PP LAB WEEK-3

DSE VI-A2 Divansh Prasad 210968140

1) Write an OpenMP program to implement Matrix multiplication.
a. Analyse the speedup and efficiency of the parallelized code.
b. Vary the size of your matrices from 200, 400, 600, 800 and 1000
and measure the runtime with one thread and four threads.
c. For each matrix size, change the number of threads from 2,4,6 and
8 and plot the speedup versus the number of threads. Compute the
efficiency.

```
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
#include <time.h>
#include <windows.h>
#define MAX_VALUE 100

void generate_matrix(int** matrix, int rows, int cols) {
    srand(time(NULL));
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = rand() % MAX_VALUE;
        }
    }
}

void matrix_multiplication_sequential(int** a, int** z, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            z[i][j] = a[i][j] * a[i][j];
        }
    }
}

void matrix_multiplication_parallel(int** a, int** z, int size,int
num_threads) {
```

```
    #pragma omp parallel for collapse(2) num_threads(num_threads)
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            z[i][j] = a[i][j] * a[i][j];
        }
    }
}

int main() {
    clock_t start, end;
    double cpu_time_used_sequential = 0;
    double cpu_time_used_parallel = 0;
    int num_threads = 1;

    printf("Matrix 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 **a = (int **)malloc(size * sizeof(int *));
            int **z = (int **)malloc(size * sizeof(int *));
            for (int i = 0; i < size; i++) {
                a[i] = (int *)malloc(size * sizeof(int));
                z[i] = (int *)malloc(size * sizeof(int));
            }

            generate_matrix(a, size, size);

            start = clock();Sleep(10);
            matrix_multiplication_sequential(a, z, size);
            end = clock();Sleep(10);
            cpu_time_used_sequential = ((double)(end - start)) /
CLOCKS_PER_SEC;

            start = clock();
            matrix_multiplication_parallel(a, z, size,num_threads);
            end = clock();
            cpu_time_used_parallel = ((double)(end - start)) /
CLOCKS_PER_SEC;
```

```c
            printf("%dx%d\t\t%d\t%.6f\t\t%.6f\t\t%.6f\t%.6f\n", size,
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);

            for (int i = 0; i < size; i++) {
                free(a[i]);
                free(z[i]);
            }
            free(a);
            free(z);
        }
    }
    return 0;
}
```

Matrix Size	Threads	Sequential Time (s)	Parallel Time (s)	Speedup	Efficiency
200x200	2	0.016000	0.000000	inf	inf
200x200	4	0.011000	0.000000	inf	inf
200x200	6	0.012000	0.001000	12.000000	2.000000
200x200	8	0.024000	0.000000	inf	inf
400x400	2	0.026000	0.000000	inf	inf
400x400	4	0.014000	0.001000	14.000000	3.500000
400x400	6	0.011000	0.001000	11.000000	1.833333
400x400	8	0.012000	0.001000	12.000000	1.500000
600x600	2	0.013000	0.001000	13.000000	6.500000
600x600	4	0.011000	0.001000	11.000000	2.750000
600x600	6	0.018000	0.001000	18.000000	3.000000
600x600	8	0.018000	0.001000	18.000000	2.250000
800x800	2	0.016000	0.002000	8.000000	4.000000
800x800	4	0.014000	0.002000	7.000000	1.750000
800x800	6	0.012000	0.002000	6.000000	1.000000
800x800	8	0.020000	0.002000	10.000000	1.250000
1000x1000	2	0.021000	0.003000	7.000000	3.500000
1000x1000	4	0.020000	0.003000	6.666667	1.666667
1000x1000	6	0.021000	0.002000	10.500000	1.750000
1000x1000	8	0.024000	0.003000	8.000000	1.000000

2) Write an OpenMP program to perform Matrix times vector multiplication. Vary the matrix and vector size and analyze the speedup and efficiency of the parallelized code.

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

#define MAX_VALUE 100

void generate_matrix(int** matrix, int rows, int cols) {
    srand(time(NULL));
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = rand() % MAX_VALUE;
        }
    }
}

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

void matrix_vector_multiplication_sequential(int** matrix, int* vector,
int* result, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        result[i] = 0;
        for (int j = 0; j < cols; j++) {
            result[i] += matrix[i][j] * vector[j];
        }
    }
}

void matrix_vector_multiplication_parallel(int** matrix, int* vector, int*
result, int rows, int cols, int num_threads) {
    #pragma omp parallel for num_threads(num_threads)
    for (int i = 0; i < rows; i++) {
```

```
        result[i] = 0;
        for (int j = 0; j < cols; j++) {
            result[i] += matrix[i][j] * vector[j];
        }
    }
}
int main() {
    clock_t start, end;
    double cpu_time_used_sequential = 0;
    double cpu_time_used_parallel = 0;
    printf("Matrix Size\tVector Size\tThreads\tSequential Time
(s)\tParallel Time (s)\tSpeedup\t\tEfficiency\n");
    for (int size = 200; size <= 1000; size += 200) {
            for (int num_threads = 2; num_threads <= 8; num_threads += 2)
{
                int **matrix = (int **)malloc(size * sizeof(int *));
                int *vector = (int *)malloc(size * sizeof(int));
                int *result_sequential = (int *)malloc(size *
sizeof(int));
                int *result_parallel = (int *)malloc(size * sizeof(int));

                for (int i = 0; i < size; i++) {
                    matrix[i] = (int *)malloc(size * sizeof(int));
                }

                generate_matrix(matrix, size, size);
                generate_vector(vector, size);

                start = clock();
                matrix_vector_multiplication_sequential(matrix, vector,
result_sequential, size, size);
                end = clock();
                cpu_time_used_sequential = ((double)(end - start)) /
CLOCKS_PER_SEC;

                start = clock();
                matrix_vector_multiplication_parallel(matrix, vector,
result_parallel, size, size, num_threads);
                end = clock();
```

```c
                cpu_time_used_parallel = ((double)(end - start)) /
CLOCKS_PER_SEC;
printf("%dx%d\t\t%dx1\t\t%d\t%.6f\t\t%.6f\t\t%.6f\t%.6f\n", size, size,
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);

                for (int i = 0; i < size; i++) {
                    free(matrix[i]);
                }
                free(matrix);
                free(vector);
                free(result_sequential);
                free(result_parallel);
            }
    }
    return 0;
}
```

Matrix Size	Vector Size	Threads	Sequential Time (s)	Parallel Time (s)	Speedup	Efficiency
200x200	200x1	2	0.000000	0.000000	-nan(ind)	-nan(ind)
200x200	200x1	4	0.000000	0.000000	-nan(ind)	-nan(ind)
200x200	200x1	6	0.000000	0.000000	-nan(ind)	-nan(ind)
200x200	200x1	8	0.000000	0.000000	-nan(ind)	-nan(ind)
400x400	400x1	2	0.001000	0.000000	inf	inf
400x400	400x1	4	0.001000	0.000000	inf	inf
400x400	400x1	6	0.000000	0.000000	-nan(ind)	-nan(ind)
400x400	400x1	8	0.001000	0.000000	inf	inf
600x600	600x1	2	0.001000	0.000000	inf	inf
600x600	600x1	4	0.001000	0.001000	1.000000	0.250000
600x600	600x1	6	0.001000	0.001000	1.000000	0.166667
600x600	600x1	8	0.000000	0.001000	0.000000	0.000000
800x800	800x1	2	0.001000	0.001000	1.000000	0.500000
800x800	800x1	4	0.001000	0.001000	1.000000	0.250000
800x800	800x1	6	0.001000	0.001000	1.000000	0.166667
800x800	800x1	8	0.001000	0.001000	1.000000	0.125000
1000x1000	1000x1	2	0.002000	0.002000	1.000000	0.500000
1000x1000	1000x1	4	0.001000	0.002000	0.500000	0.125000
1000x1000	1000x1	6	0.002000	0.002000	1.000000	0.166667
1000x1000	1000x1	8	0.002000	0.001000	2.000000	0.250000

3) Write an OpenMp program to read a matrix A of size 5x5. It produces a resultant matrix B of size 5x5. It sets all the principal diagonal elements of B matrix with 0. It replaces each row elements in the B matrix in the following manner. If the element is below the principal diagonal it replaces it with the maximum value of the row in the A matrix having the same row number of B. If the element is above the principal diagonal it replaces it with the minimum value of the row in the A matrix having the same row number of B. Analyze the speedup and efficiency of the parallelized code.

```
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
#include <time.h>
#include <windows.h>
#define MAX_VALUE 100

void generate_matrix(int** matrix, int rows, int cols) {
    srand(time(NULL));
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = rand() % MAX_VALUE;
        }
    }
}
void processMatrixparallel(int** a, int** z, int size,int num) {
    #pragma omp parallel for nested(2) num_threads(num)
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (j == i) {
                z[i][j] = 0;
            } else if (j > i) {
                int maxVal = a[i][0];
                for (int k = 1; k < size; k++) {
                    if (a[i][k] > maxVal) {
                        maxVal = a[i][k];
                    }
                }
                z[i][j] = maxVal;
            } else {
                int minVal = a[i][0];
                for (int k = 1; k < size; k++) {
                    if (a[i][k] < minVal) {
```

```
                        minVal = a[i][k];
                    }
                }
                z[i][j] = minVal;
            }
        }
    }
}
void processMatrixsequential(int** a, int** z, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (j == i) {
                z[i][j] = 0;
            } else if (j > i) {
                int maxVal = a[i][0];
                for (int k = 1; k < size; k++) {
                    if (a[i][k] > maxVal) {
                        maxVal = a[i][k];
                    }
                }
                z[i][j] = maxVal;
            } else {
                int minVal = a[i][0];
                for (int k = 1; k < size; k++) {
                    if (a[i][k] < minVal) {
                        minVal = a[i][k];
                    }
                }
                z[i][j] = minVal;
            }
        }
    }
}
int main() {
    clock_t start, end;
    double cpu_time_used_sequential = 0;
    double cpu_time_used_parallel = 0;
    int num_threads = 1;
```

```
    printf("Matrix 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 **a = (int **)malloc(size * sizeof(int *));
            int **z = (int **)malloc(size * sizeof(int *));
            for (int i = 0; i < size; i++) {
                a[i] = (int *)malloc(size * sizeof(int));
                z[i] = (int *)malloc(size * sizeof(int));
            }

            generate_matrix(a, size, size);

            start = clock();Sleep(10);
            processMatrixsequential(a, z, size);
            end = clock();Sleep(10);
            cpu_time_used_sequential = ((double)(end - start)) /
CLOCKS_PER_SEC;

            start = clock();
            processMatrixparallel(a, z, size,num_threads);
            end = clock();
            cpu_time_used_parallel = ((double)(end - start)) /
CLOCKS_PER_SEC;

            printf("%dx%d\t\t%d\t%.6f\t\t%.6f\t\t%.6f\t%.6f\n", size,
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);

            for (int i = 0; i < size; i++) {
                free(a[i]);
                free(z[i]);
            }
            free(a);
            free(z);
        }
    }
```

```
    return 0;
}
```

Matrix Size	Threads	Sequential Time (s)	Parallel Time (s)	Speedup	Efficiency
200x200	2	0.023000	0.009000	2.555556	1.277778
200x200	4	0.031000	0.009000	3.444444	0.861111
200x200	6	0.024000	0.009000	2.666667	0.444444
200x200	8	0.026000	0.009000	2.888889	0.361111
400x400	2	0.079000	0.068000	1.161765	0.580882
400x400	4	0.078000	0.066000	1.181818	0.295455
400x400	6	0.095000	0.066000	1.439394	0.239899
400x400	8	0.082000	0.066000	1.242424	0.155303
600x600	2	0.233000	0.218000	1.068807	0.534404
600x600	4	0.226000	0.217000	1.041475	0.260369
600x600	6	0.232000	0.216000	1.074074	0.179012
600x600	8	0.219000	0.219000	1.000000	0.125000
800x800	2	0.513000	0.505000	1.015842	0.507921
800x800	4	0.498000	0.514000	0.968872	0.242218
800x800	6	0.506000	0.547000	0.925046	0.154174
800x800	8	0.527000	0.510000	1.033333	0.129167
1000x1000	2	0.980000	0.996000	0.983936	0.491968
1000x1000	4	0.976000	0.985000	0.990863	0.247716
1000x1000	6	0.965000	0.982000	0.982688	0.163781
1000x1000	8	0.982000	1.012000	0.970356	0.121294

4) Write a parallel program using OpenMP that reads a matrix of size MxN and produce an output matrix B of same size such that it replaces all the non-border elements of A with its equivalent 1's complement and remaining elements same as matrix A. Also produce a matrix D as shown below.

```
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
#include <time.h>
#include <windows.h>
#define MAX_VALUE 100

void generate_matrix(int** matrix, int rows, int cols) {
    srand(time(NULL));
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = rand() % MAX_VALUE;
        }
    }
}
int onesComplement(int num) {
    return ~num;
}
void processMatrixsequential(int **a, int **b, int **d,int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (i != 0 && j != 0 && i != size - 1 && j != size - 1) {
                b[i][j] = onesComplement(a[i][j]);
            } else {
                b[i][j] = a[i][j];
                d[i][j] = a[i][j];
            }
        }
    }
}
void processMatrixparallel(int **a, int **b, int **d,int size,int num) {
    #pragma omp for collapsed(2) num_threads(num)
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (i != 0 && j != 0 && i != size - 1 && j != size - 1) {
```

```c
                b[i][j] = onesComplement(a[i][j]);
            } else {
                b[i][j] = a[i][j];
                d[i][j] = a[i][j];
            }
        }
    }
}
int main() {
    clock_t start, end;
    double cpu_time_used_sequential = 0;
    double cpu_time_used_parallel = 0;
    int num_threads = 1;
    printf("Matrix 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 **a = (int **)malloc(size * sizeof(int *));
            int **b = (int **)malloc(size * sizeof(int *));
            int **d = (int **)malloc(size * sizeof(int *));
            for (int i = 0; i < size; i++) {
                a[i] = (int *)malloc(size * sizeof(int));
                b[i] = (int *)malloc(size * sizeof(int));
                d[i] = (int *)malloc(size * sizeof(int));
            }

            generate_matrix(a, size, size);

            start = clock();Sleep(10);
            processMatrixsequential(a,b,d, size);
            end = clock();Sleep(10);
            cpu_time_used_sequential = ((double)(end - start)) /
CLOCKS_PER_SEC;

            start = clock();
            processMatrixparallel(a, b,d, size,num_threads);
            end = clock();
            cpu_time_used_parallel = ((double)(end - start)) /
CLOCKS_PER_SEC;
```

```c
            printf("%dx%d\t\t%d\t%.6f\t\t%.6f\t\t%.6f\t%.6f\n", size,
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);
            for (int i = 0; i < size; i++) {
                free(a[i]);
                free(b[i]);
                free(d[i]);
            }
            free(a);
            free(b);
            free(d);
        }
    }
    return 0;
}
```

Matrix Size	Threads	Sequential Time (s)	Parallel Time (s)	Speedup	Efficiency
200x200	2	0.016000	0.000000	inf	inf
200x200	4	0.015000	0.000000	inf	inf
200x200	6	0.010000	0.000000	inf	inf
200x200	8	0.013000	0.000000	inf	inf
400x400	2	0.016000	0.001000	16.000000	8.000000
400x400	4	0.012000	0.001000	12.000000	3.000000
400x400	6	0.012000	0.001000	12.000000	2.000000
400x400	8	0.012000	0.001000	12.000000	1.500000
600x600	2	0.012000	0.002000	6.000000	3.000000
600x600	4	0.020000	0.001000	20.000000	5.000000
600x600	6	0.015000	0.002000	7.500000	1.250000
600x600	8	0.012000	0.002000	6.000000	0.750000
800x800	2	0.025000	0.003000	8.333333	4.166667
800x800	4	0.026000	0.003000	8.666667	2.166667
800x800	6	0.021000	0.002000	10.500000	1.750000
800x800	8	0.020000	0.003000	6.666667	0.833333
1000x1000	2	0.029000	0.004000	7.250000	3.625000
1000x1000	4	0.022000	0.004000	5.500000	1.375000
1000x1000	6	0.025000	0.004000	6.250000	1.041667
1000x1000	8	0.024000	0.004000	6.000000	0.750000

5) Write a parallel program in OpenMP to reverse the digits of the following integer array of size 9. Initialise the input array to the following values:

a. Input array: 18, 523, 301, 1234, 2, 14, 108, 150, 1928

b. Output array: 81, 325, 103, 4321, 2, 41, 801, 51, 8291

```
 #include <stdio.h>
#include <omp.h>
#include <time.h>
#include <windows.h>
int main(){
    clock_t start, end;
    double cpu_time_used=0;
    int rev=0;
    int X[9]={18, 523, 301, 1234, 2, 14, 108, 150, 1928};
    printf("Input Array: 18\t523\t301\t1234\t2\t14\t108\t150\t1928\nOutpt Array: ");
    start = clock();
    Sleep(10);
    #pragma omp parallel for reduction(*:rev)
    for (int j=0;j<9;j++){
            for (int k=X[j];k>0;k=k/10){
                rev=(rev*10)+(k%10);
                }
            printf("%d\t",rev);
            rev=0;
            }
        end = clock();
        cpu_time_used=cpu_time_used +((double) (end - start)) / CLOCKS_PER_SEC;
        printf("\n\nTime taken to reverse elements of entire array: %0.3f\n",cpu_time_used);
    return 0;
    }
```

```
Input Array: 18 523     301     1234    2       14      108     150     1928
Outpt Array: 81 325     103     4321    2       41      801     51      8291

Time taken to reverse elements of entire array: 0.015
```