**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;

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);

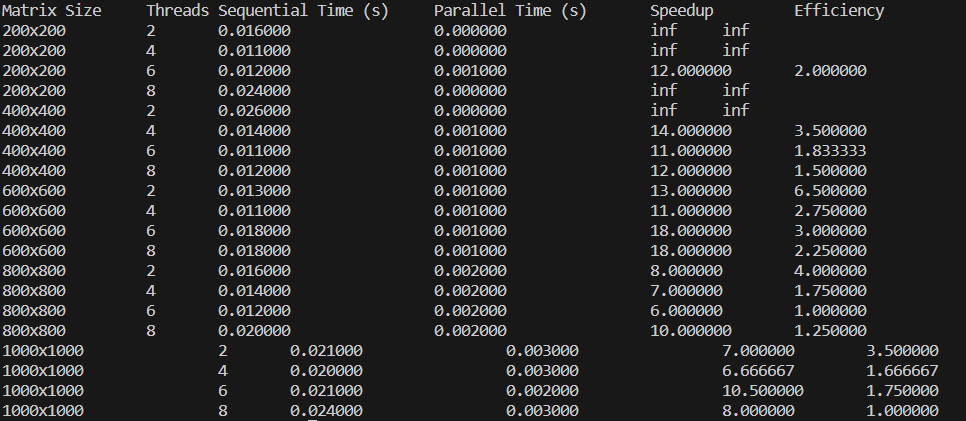
free(z);

}

}

return 0;

}



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();

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);

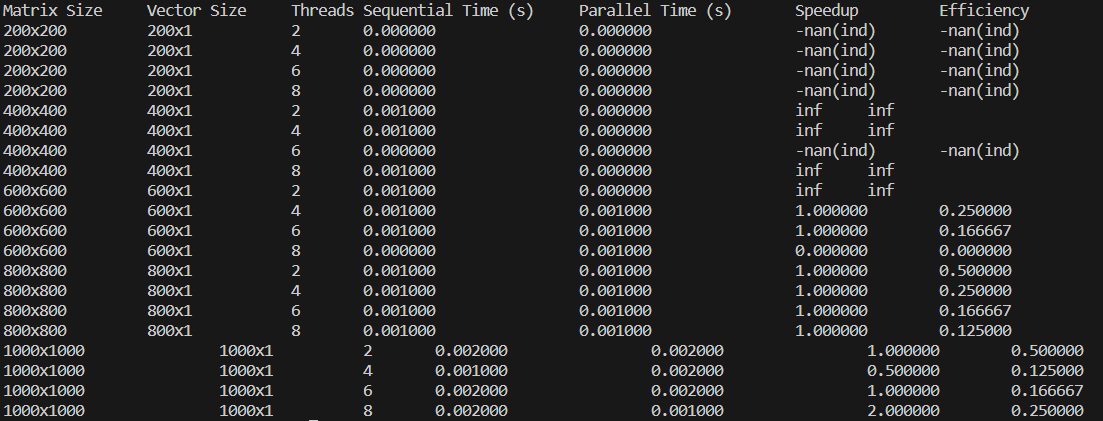
free(result\_parallel);

}

}

return 0;

}



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);

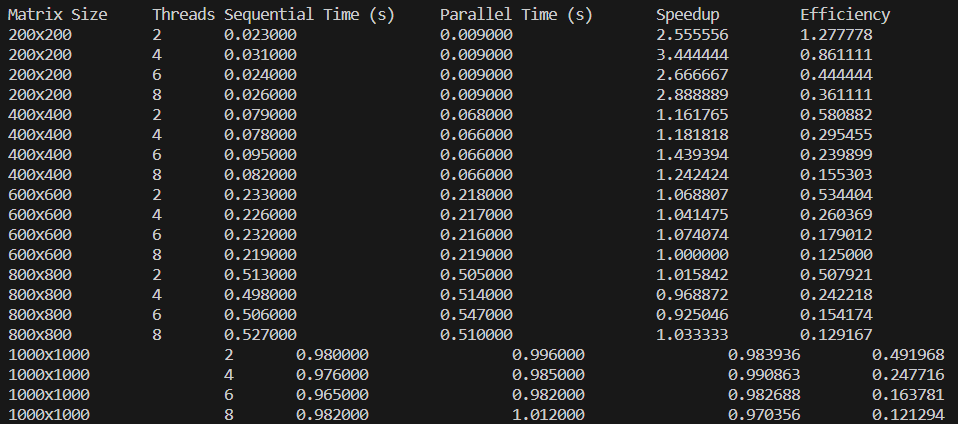
free(z);

}

}

return 0;

}



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) {

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;

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);

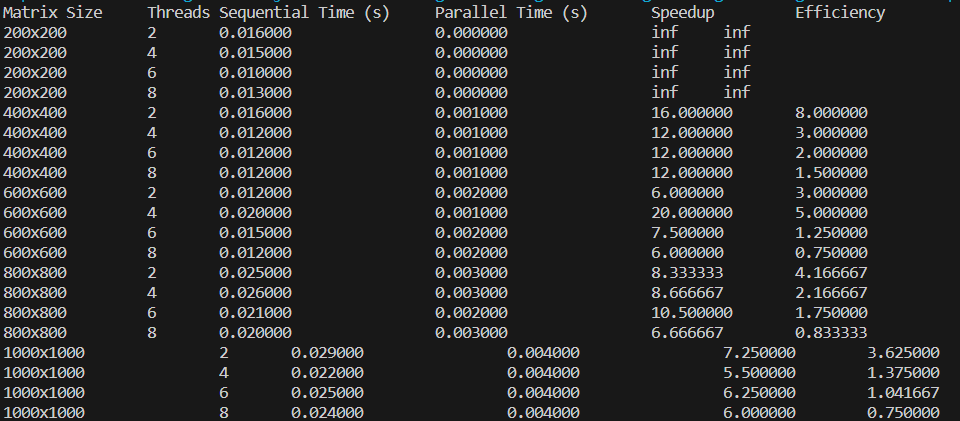
free(d);

}

}

return 0;

}



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;

}

