**20BCE1550**

**Samridh Anand Paatni**

**CSE4001 Lab 4**

**Sections**

**Q1.**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

void function1() {

    // int n = 100000;

    int n = 16;

    int \*nums = (int \*)calloc(n, sizeof(int));

    for (int i = 0; i < n; i++) {

        nums[i] = rand() % 100;

    }

    int i = 0;

    int local\_min\_1, local\_min\_2, local\_min\_3, local\_min\_4;

    int min = 100;

    #pragma omp parallel shared(nums, n, min) private(i, local\_min\_1, local\_min\_2, local\_min\_3, local\_min\_4) num\_threads(4)

    {

        #pragma omp sections

        {

            min = 100;

            #pragma omp section

            {

                local\_min\_1 = 100;

                for (i = 0; i < n/4; i++) {

                    if (nums[i] < local\_min\_1) local\_min\_1 = nums[i];

                }

                #pragma omp critical

                {

                    if (local\_min\_1 < min) min = local\_min\_1;

                }

            }

            #pragma omp section

            {

                local\_min\_2 = 100;

                for (i = n/4; i < 2 \* n/4; i++) {

                    if (nums[i] < local\_min\_2) local\_min\_2 = nums[i];

                }

                #pragma omp critical

                {

                    if (local\_min\_2 < min) min = local\_min\_2;

                }

            }

            #pragma omp section

            {

                local\_min\_3 = 100;

                for (i = 2 \* n/4; i < 3 \* n/4; i++) {

                    if (nums[i] < local\_min\_3) local\_min\_3 = nums[i];

                }

                #pragma omp critical

                {

                    if (local\_min\_3 < min) min = local\_min\_3;

                }

            }

            #pragma omp section

            {

                local\_min\_4 = 100;

                for (i = 3 \* n/4; i < n; i++) {

                    if (nums[i] < local\_min\_4) local\_min\_4 = nums[i];

                }

                #pragma omp critical

                {

                    if (local\_min\_4 < min) min = local\_min\_4;

                }

            }

        }

    }

        for (i = 0; i < n; i++) {

            printf(

                "%s%d%s",

                i == 0 ? "[" : " ",

                nums[i],

                i == n - 1 ? "]\n" : ","

            );

        }

        printf("\n%d is the minimum element", min);

        free(nums);

}

int main() {

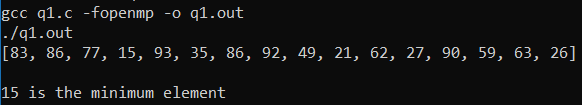
    function1();

    printf("\n");

    return 0;

}

**Output:**

****

**Q2.**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

typedef int \*\* matrix;

#define N 1100 // the size of the matrices

void multiply(matrix A, matrix B, matrix ans) {

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

            for (int k = 0; k < N; k++) {

                ans[i][j] += A[i][k] \* B[k][j];

            }

        }

    }

}

// case 1 = only the outermost loop is parallelized

void multiplyCase1(matrix A, matrix B, matrix ans, int numThreads) {

    printf("The outermost loop is parallelized\n");

    int i = 0;

    #pragma omp parallel for shared(numThreads, A, B, ans) private(i)

    for (i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

            for (int k = 0; k < N; k++) {

                ans[i][j] += A[i][k] \* B[k][j];

            }

        }

    }

}

void multiplyCase2(matrix A, matrix B, matrix ans, int numThreads) {

    printf("The outer 2 loops are parallelized\n");

    int i = 0, j = 0;

    #pragma omp parallel for shared(numThreads, A, B, ans) private(i)

    for (i = 0; i < N; i++) {

        #pragma omp parallel for

        for (j = 0; j < N; j++) {

            for (int k = 0; k < N; k++) {

                ans[i][j] += A[i][k] \* B[k][j];

            }

        }

    }

}

void multiplyCase3(matrix A, matrix B, matrix ans, int numThreads) {

    printf("All loops are parallelized\n");

    int i = 0, j = 0, k = 0;

    #pragma omp parallel for shared(numThreads, A, B, ans) private(i)

    for (i = 0; i < N; i++) {

        #pragma omp parallel for

        for (j = 0; j < N; j++) {

            #pragma omp parallel for

            for (k = 0; k < N; k++) {

                ans[i][j] += A[i][k] \* B[k][j];

            }

        }

    }

}

void multiplyCase4(matrix A, matrix B, matrix ans, int numThreads) {

    printf("All loops are parallelized\n");

    int i = 0, j = 0, k = 0;

    #pragma omp parallel for shared(numThreads, A, B, ans) private(i) collapse(3)

    for (i = 0; i < N; i++) {

        for (j = 0; j < N; j++) {

            for (k = 0; k < N; k++) {

                ans[i][j] += A[i][k] \* B[k][j];

            }

        }

    }

}

void print(matrix A) {

    for (int i = 0; i < N; i++) {

        printf("%s", i == 0 ? "[\n " : " ");

        for (int j = 0; j < N; j++) {

            printf(

                "%s%d%s",

                j == 0 ? "[" : " ",

                A[i][j],

                j == N - 1 ? "]" : ","

            );

        }

        printf("%s", i == N - 1 ? "\n]\n" : ",\n");

    }

}

int main(int argc, char \*argv[]) {

    int numThreads = atoi(argv[1]);

    int type = atoi(argv[2]);

    printf("multiplying 2 %dx%d matrices using %d threads and case %d:\n", N, N, numThreads, type);

    matrix A, B, C;

    A = (int \*\*)calloc(N, sizeof(int\*));

    B = (int \*\*)calloc(N, sizeof(int\*));

    C = (int \*\*)calloc(N, sizeof(int\*));

    for (int i = 0; i < N; i++) {

        A[i] = (int \*) calloc(N, sizeof(int));

        B[i] = (int \*) calloc(N, sizeof(int));

        C[i] = (int \*) calloc(N, sizeof(int));

        for (int j = 0; j < N; j++) {

            A[i][j] = rand() % 10;

            B[i][j] = rand() % 10;

            C[i][j] = 0;

        }

    }

    // print(A);

    // print(B);

    double t = omp\_get\_wtime();

    switch (type)

    {

    case 0:

        multiply(A, B, C);

        break;

    case 1:

        multiplyCase1(A, B, C, numThreads);

        break;

    case 2:

        multiplyCase2(A, B, C, numThreads);

        break;

    case 3:

        multiplyCase3(A, B, C, numThreads);

        break;

    case 4:

        multiplyCase4(A, B, C, numThreads);

        break;

    default:

    }

    t = omp\_get\_wtime() - t;

    // print(C);

    printf("took %f seconds\n", t);

    for (int i = 0; i < N; i++) {

        free(A[i]);

        free(B[i]);

    }

    free(A);

    free(B);

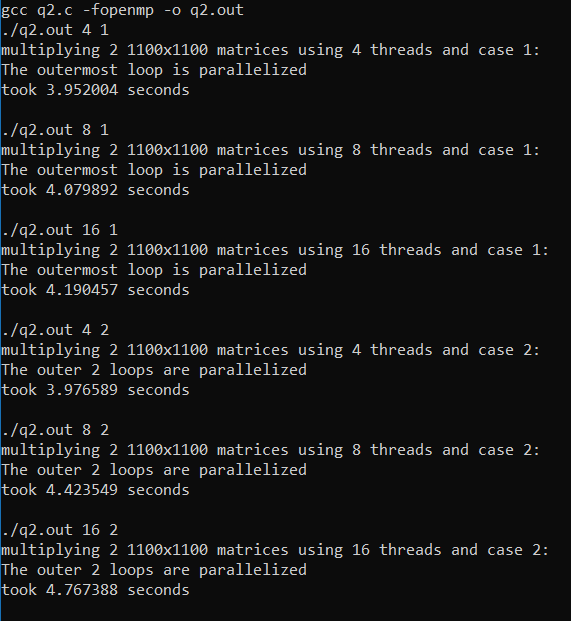
    free(C);

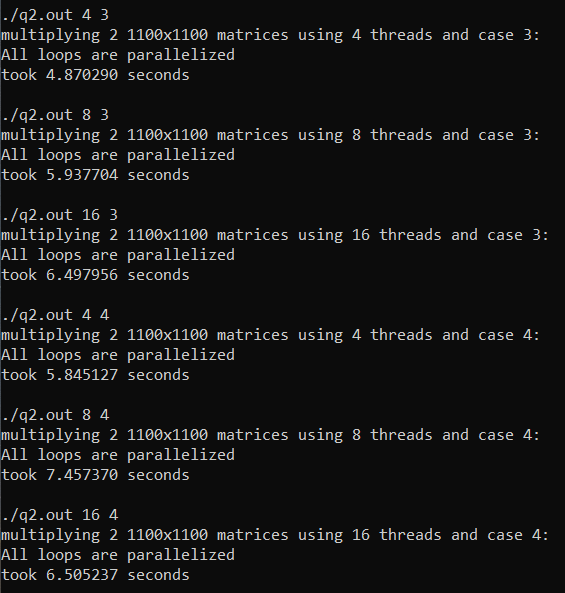
    printf("\n");

    return 0;

}

**Output:**

****

****