Abdul Mohiz Khan Tareen (FA20 – BCS – 001)

Muhammad Taha Malik (FA20 – BCS – 015)

Talal Khan (FA20 – BCS – 029)

Lab Mid Term

Code: Serial Execution

```
#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <time.h>


#define SIZE 250

#define EVEN_COUNT 100


void initialize_array(int arr[SIZE][SIZE]) {

    for (int i = 0; i < SIZE; i++) {

        for (int j = 0; j < SIZE; j++) {

            arr[i][j] = rand() % 1000; // Random values between 0 and 999

        }

    }

}


void find_even_numbers(int A[SIZE][SIZE], int B[SIZE][SIZE], int C[SIZE][SIZE], 
int D[EVEN_COUNT]) {

    int count = 0;


    for (int i = 0; i < SIZE && count < EVEN_COUNT; i++) {

        for (int j = 0; j < SIZE && count < EVEN_COUNT; j++) {

            if (A[i][j] % 2 == 0 && count < EVEN_COUNT) {

                D[count++] = A[i][j];

            }
            if (B[i][j] % 2 == 0 && count < EVEN_COUNT) {
                D[count++] = B[i][j];
            }
            if (C[i][j] % 2 == 0 && count < EVEN_COUNT) {
                D[count++] = C[i][j];
            }
        }
    }
}

void print_array(int arr[EVEN_COUNT]) {
    for (int i = 0; i < EVEN_COUNT; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];
    int D[EVEN_COUNT];

    srand(time(NULL)); // Seed for random number generation

    initialize_array(A);
    initialize_array(B);
    initialize_array(C);

    find_even_numbers(A, B, C, D);

    print_array(D);
    return 0;
}
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

#define SIZE 250
#define EVEN_COUNT 100

void initialize_array(int arr[SIZE][SIZE]) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            arr[i][j] = rand() % 1000; // Random values between 0 and 999
        }
    }
}

void find_even_numbers_parallel(int A[SIZE][SIZE], int B[SIZE][SIZE], int
C[SIZE][SIZE], int D[EVEN_COUNT], int threads) {
    int count = 0;

    #pragma omp parallel for num_threads(threads) collapse(2) shared(count)
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            // Check if we have already found enough even numbers
            if (count >= EVEN_COUNT) {
                // Exit the loop safely by using a flag
                i = SIZE;
                j = SIZE;
                continue;
            }
            #pragma omp critical
            {
                if (A[i][j] % 2 == 0 && count < EVEN_COUNT) {
                    D[count++] = A[i][j];
                }
                if (B[i][j] % 2 == 0 && count < EVEN_COUNT) {
                    D[count++] = B[i][j];
                }
                if (C[i][j] % 2 == 0 && count < EVEN_COUNT) {
                    D[count++] = C[i][j];
                }
            }
        }
    }
}

void measure_execution_time(int threads, int A[SIZE][SIZE], int B[SIZE][SIZE],
int C[SIZE][SIZE], int D[EVEN_COUNT]) {
    double start, end;

    start = omp_get_wtime();
    find_even_numbers_parallel(A, B, C, D, threads);
    end = omp_get_wtime();

    printf("Threads: %d, Time: %f seconds\n", threads, end - start);
}

void print_array(int arr[EVEN_COUNT]) {
    for (int i = 0; i < EVEN_COUNT; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];
    int D[EVEN_COUNT];

    srand(time(NULL)); // Seed for random number generation

    initialize_array(A);
    initialize_array(B);
    initialize_array(C);

    measure_execution_time(1, A, B, C, D);   // Serial execution
    measure_execution_time(2, A, B, C, D);   // Parallel execution with 2 threads
    measure_execution_time(4, A, B, C, D);   // Parallel execution with 4 threads
    measure_execution_time(8, A, B, C, D);   // Parallel execution with 8 threads
    measure_execution_time(12, A, B, C, D);  // Parallel execution with 12 threads
    measure_execution_time(16, A, B, C, D);  // Parallel execution with 16 threads
    measure_execution_time(24, A, B, C, D);  // Parallel execution with 24 threads

    print_array(D);

    return 0;
}
```

Code: Execution with Threads

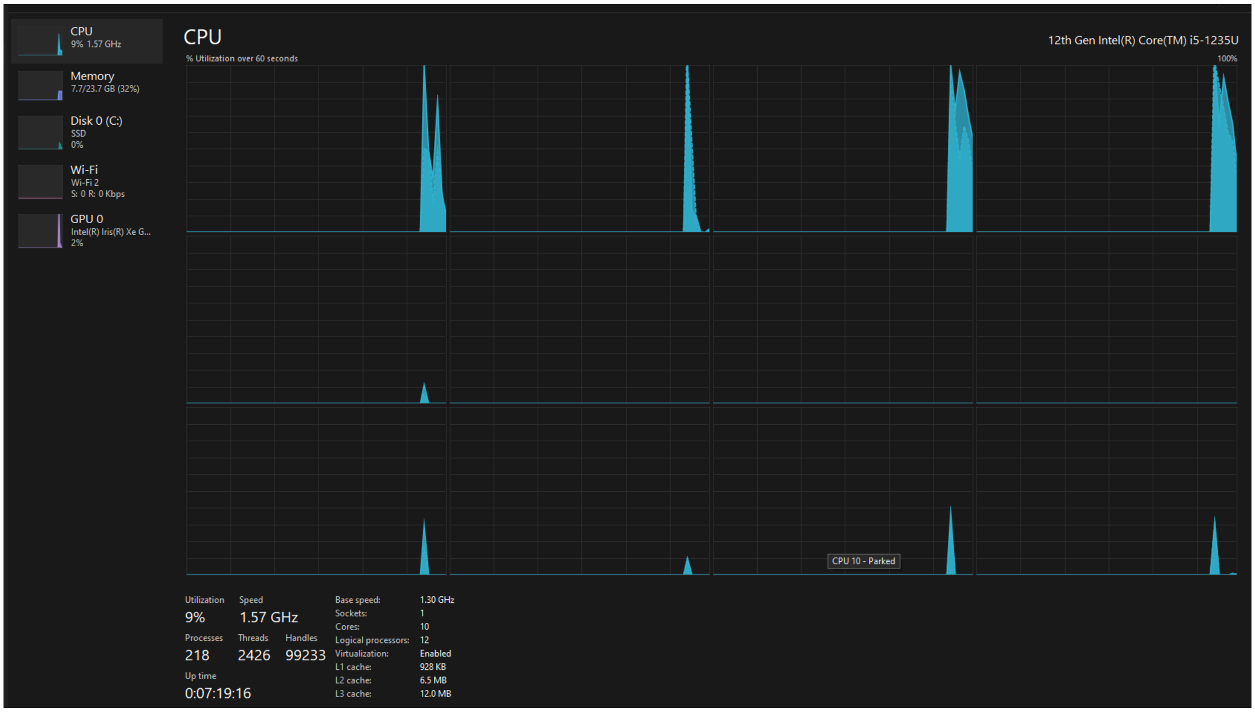
```
#include <stdio.h>
```

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

#define SIZE 250
#define EVEN_COUNT 100

void initialize_array(int arr[SIZE][SIZE]) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            arr[i][j] = rand() % 1000; // Random values between 0 and 999
        }
    }
}

void find_even_numbers_parallel(int A[SIZE][SIZE], int B[SIZE][SIZE], int C[SIZE][SIZE], int D[EVEN_COUNT], int threads) {
    int count = 0;

    #pragma omp parallel for num_threads(threads) collapse(2) shared(count)
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            // Check if we have already found enough even numbers
            if (count >= EVEN_COUNT) {
                // Exit the loop safely by using a flag
                i = SIZE;
                j = SIZE;
                continue;
            }

            #pragma omp critical
            {
                if (A[i][j] % 2 == 0 && count < EVEN_COUNT) {
                    D[count++] = A[i][j];
                }
                if (B[i][j] % 2 == 0 && count < EVEN_COUNT) {
                    D[count++] = B[i][j];
                }
                if (C[i][j] % 2 == 0 && count < EVEN_COUNT) {
                    D[count++] = C[i][j];
                }
            }
        }
    }
}

void measure_execution_time(int threads, int A[SIZE][SIZE], int B[SIZE][SIZE], int
C[SIZE][SIZE], int D[EVEN_COUNT]) {
    double start, end;

    start = omp_get_wtime();
    find_even_numbers_parallel(A, B, C, D, threads);
    end = omp_get_wtime();

    printf("Threads: %d, Time: %f seconds\n", threads, end - start);
}

void print_array(int arr[EVEN_COUNT]) {
    for (int i = 0; i < EVEN_COUNT; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];
    int D[EVEN_COUNT];
    srand(time(NULL)); // Seed for random number generation

    initialize_array(A);
    initialize_array(B);
    initialize_array(C);

    measure_execution_time(1, A, B, C, D);   // Serial execution
    measure_execution_time(2, A, B, C, D);   // Parallel execution with 2 threads
    measure_execution_time(4, A, B, C, D);   // Parallel execution with 4 threads
    measure_execution_time(8, A, B, C, D);   // Parallel execution with 8 threads
    measure_execution_time(12, A, B, C, D);  // Parallel execution with 12 threads
    measure_execution_time(16, A, B, C, D);  // Parallel execution with 16 threads
    measure_execution_time(24, A, B, C, D);  // Parallel execution with 24 threads

    print_array(D);

    return 0;
}
```

Code: Optimizing Code with Scheduling Methods

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

#define SIZE 250
#define EVEN_COUNT 100

void initialize_array(int arr[SIZE][SIZE]) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            arr[i][j] = rand() % 1000; // Random values between 0 and 999
        }
    }
}

void print_array(int arr[EVEN_COUNT]) {
    for (int i = 0; i < EVEN_COUNT; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}


void find_even_numbers_parallel_optimized(int A[SIZE][SIZE], int B[SIZE][SIZE], int C[SIZE][SIZE], int D[EVEN_COUNT], int threads, omp_sched_t schedule) {
    int count = 0;

    omp_set_schedule(schedule, 0);
    #pragma omp parallel for num_threads(threads) schedule(runtime) collapse(2) shared(count)
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            // Check if we have already found enough even numbers
            if (count >= EVEN_COUNT) {
                // Exit the loop safely by using a flag
                i = SIZE;
                j = SIZE;
                continue;
            }

            #pragma omp critical
            {
                if (A[i][j] % 2 == 0 && count < EVEN_COUNT) {
                    D[count++] = A[i][j];
                }
                if (B[i][j] % 2 == 0 && count < EVEN_COUNT) {
                    D[count++] = B[i][j];
                }
                if (C[i][j] % 2 == 0 && count < EVEN_COUNT) {
                    D[count++] = C[i][j];
                }
            }
        }
    }
}

void measure_execution_time_with_scheduling(int threads, int A[SIZE][SIZE], int
B[SIZE][SIZE], int C[SIZE][SIZE], int D[EVEN_COUNT]) {
    double start, end;

    omp_set_schedule(omp_sched_static, 0);
    start = omp_get_wtime();
    find_even_numbers_parallel_optimized(A, B, C, D, threads, omp_sched_static);
    end = omp_get_wtime();
    printf("Static Schedule, Threads: %d, Time: %f seconds\n", threads, end - start);

    omp_set_schedule(omp_sched_dynamic, 0);
    start = omp_get_wtime();
    find_even_numbers_parallel_optimized(A, B, C, D, threads, omp_sched_dynamic);
    end = omp_get_wtime();
    printf("Dynamic Schedule, Threads: %d, Time: %f seconds\n", threads, end - start);

    omp_set_schedule(omp_sched_guided, 0);
    start = omp_get_wtime();
    find_even_numbers_parallel_optimized(A, B, C, D, threads, omp_sched_guided);
    end = omp_get_wtime();
    printf("Guided Schedule, Threads: %d, Time: %f seconds\n", threads, end - start);
}

int main() {
    int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];
    int D[EVEN_COUNT];

    srand(time(NULL)); // Seed for random number generation

    initialize_array(A);
    initialize_array(B);
    initialize_array(C);

    measure_execution_time_with_scheduling(8, A, B, C, D);

    print_array(D);

    return 0;
}
```

System Specifications:



```
Utilization   Speed              Base speed:           1.30 GHz
9%            1.57 GHz           Sockets:              1
                                 Cores:                10
Processes     Threads   Handles  Logical processors:   12
218           2426      99233    Virtualization:       Enabled
                                 L1 cache:             928 KB
Up time                          L2 cache:             6.5 MB
0:07:19:16                       L3 cache:             12.0 MB
```