****

**Name: Nabiha Mahmood**

**Reg no: 22-NTU-CS-1215**

**Department: Computer Science**

**Class: BSCS-6B**

**Course: Parallel and Distributed Computing**

**Assignment no: 01**

**Submitted to: Sir Nasir Mahmood**

**Submission date: 11-03-2025**

## Matrix Multiplication Sequentially:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define SIZE 1000  // Matrix size

int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE]; // Declare matrices globally to avoid stack overflow issues

// Function to fill matrices with random values

void fill\_matrices() {

    for (int i = 0; i < SIZE; i++) {

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

            A[i][j] = rand() % 10;  // Random values between 0-9

            B[i][j] = rand() % 10;

            C[i][j] = 0;  // Initialize result matrix to 0

        }

    }

}

// Function for sequential matrix multiplication

void multiply() {

    for (int i = 0; i < SIZE; i++) {

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

            int sum = 0;

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

                sum += A[i][k] \* B[k][j];  // Multiply and sum up

            }

            C[i][j] = sum;

        }

    }

}

int main() {

    fill\_matrices();  // Fill matrices with random values

    printf("== Running Sequential Multiplication ==\n");

    double total\_time = 0;  // Store total execution time for averaging

    for (int r = 0; r < 10; r++) {  // Run 10 times

        clock\_t start = clock();  // Start time

        multiply();

        clock\_t end = clock();  // End time

        double time\_taken = (double)(end - start) / CLOCKS\_PER\_SEC;  // Convert to seconds

        total\_time += time\_taken;

        printf("Run %d Time: %.6f sec\n", r + 1, time\_taken);

    }

    printf("Average Time: %.6f sec\n", total\_time / 10);  // Print average execution time

    return 0;

}

Sequential Multiplication:  
This method performs matrix multiplication in a step-by-step manner using only one thread (single process). It goes through each row and column, multiplying values and adding them up to store the final result. Since there is no parallelism, the entire workload is handled by a single core, making it the slowest method. This approach works fine for small matrices but becomes very slow for larger ones. The more the matrix size increases, the more time it takes to complete.

## Matrix Multiplication with Static Scheduling:

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define SIZE 1000   // Size of the matrices (1000x1000)

#define CHUNK 250   // Chunk size

// Declaring matrices globally to prevent stack overflow issues

int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];

// Function to fill matrices A and B with random numbers and set C to zero

void fillMatrices() {

    for (int i = 0; i < SIZE; i++) {

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

            A[i][j] = rand() % 10;  // Fill matrix A with random numbers (0-9)

            B[i][j] = rand() % 10;  // Fill matrix B with random numbers (0-9)

            C[i][j] = 0;            // Initialize matrix C with zeros

        }

    }

}

// Function to multiply matrices using OpenMP with static scheduling

void multiplyStatic(int threads) {

    #pragma omp parallel for num\_threads(threads) schedule(static, CHUNK)

    for (int i = 0; i < SIZE; i++) {

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

            int sum = 0;

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

                sum += A[i][k] \* B[k][j];  // Multiply and calculate the sum

            }

            C[i][j] = sum;  // Store the result in matrix C

        }

    }

}

int main() {

    fillMatrices();  // Fill the matrices with random values

    int threadOptions[] = {1, 4, 8};  // Test with 1, 4, and 8 threads

    printf(" Running with Static Scheduling \n");

    // Try different numbers of threads

    for (int t = 0; t < 3; t++) {

        int threads = threadOptions[t];

        printf("\nUsing %d threads (Static Scheduling)\n", threads);

        double totalTime = 0;

        // Run the multiplication 10 times to get an average time

        for (int run = 0; run < 10; run++) {

            double start = omp\_get\_wtime();  // Start timer

            multiplyStatic(threads);  // Multiply matrices

            double end = omp\_get\_wtime();  // Stop timer

            double executionTime = end - start;

            totalTime += executionTime;  // Add to total time

            printf("Run %d Time: %.6f sec\n", run + 1, executionTime);

        }

        // Print the average execution time over 10 runs

        printf("Average Time: %.6f sec\n", totalTime / 10);

    }

    return 0;

}

Static Scheduling (OpenMP):  
In this method, we use **multiple threads** (workers) to divide the matrix into **equal chunks** for parallel processing. Each thread gets a fixed portion of the work and processes it simultaneously. This approach works well when all parts of the matrix take **roughly the same time** to compute. However, if some parts take longer, the faster threads might **sit idle,** waiting for the others to finish. It is **efficient for smaller matrices** or problems with evenly distributed workloads

## Matrix Multiplication with Dynamic Scheduling:

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define SIZE 1000   // Size of the matrices (1000x1000)

#define CHUNK 250   // Chunk size

// Declaring matrices globally to prevent stack overflow issues

int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];

// Function to fill matrices A and B with random numbers and set C to zero

void fillMatrices() {

    for (int i = 0; i < SIZE; i++) {

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

            A[i][j] = rand() % 10;  // Fill matrix A with random numbers (0-9)

            B[i][j] = rand() % 10;  // Fill matrix B with random numbers (0-9)

            C[i][j] = 0;            // Initialize matrix C with zeros

        }

    }

}

// Function to multiply matrices using OpenMP with dynamic scheduling

void multiplyDynamic(int threads) {

    #pragma omp parallel for num\_threads(threads) schedule(dynamic, CHUNK)

    for (int i = 0; i < SIZE; i++) {

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

            int sum = 0;

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

                sum += A[i][k] \* B[k][j];  // Multiply and calculate the sum

            }

            C[i][j] = sum;  // Store the result in matrix C

        }

    }

}

int main() {

    fillMatrices();

    int threadOptions[] = {1, 4, 8};  // Test with 1, 4, and 8 threads

    printf("Running with Dynamic Scheduling:\n");

    for (int t = 0; t < 3; t++) {

        int threads = threadOptions[t];

        printf("\nUsing %d threads (Dynamic Scheduling)\n", threads);

        double totalTime = 0;  // Keep track of total execution time

        // Run the multiplication 10 times to get an average time

        for (int run = 0; run < 10; run++) {

            double start = omp\_get\_wtime();  // Start timer

            multiplyDynamic(threads);  // Multiply matrices

            double end = omp\_get\_wtime();  // Stop timer

            double executionTime = end - start;

            totalTime += executionTime;  // Add to total time

            printf("Run %d Time: %.6f sec\n", run + 1, executionTime);

        }

        // Print the average execution time over 10 runs

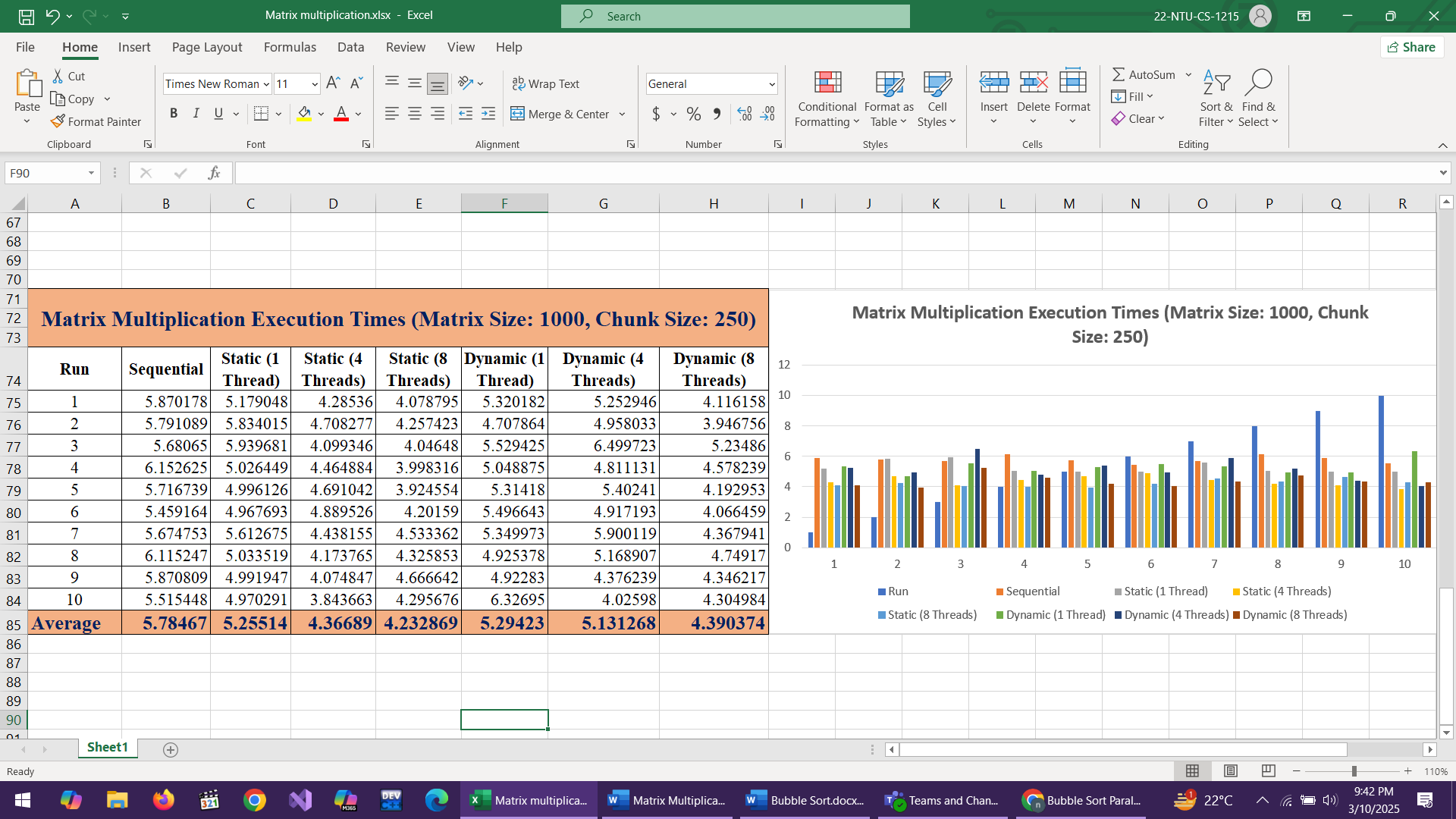
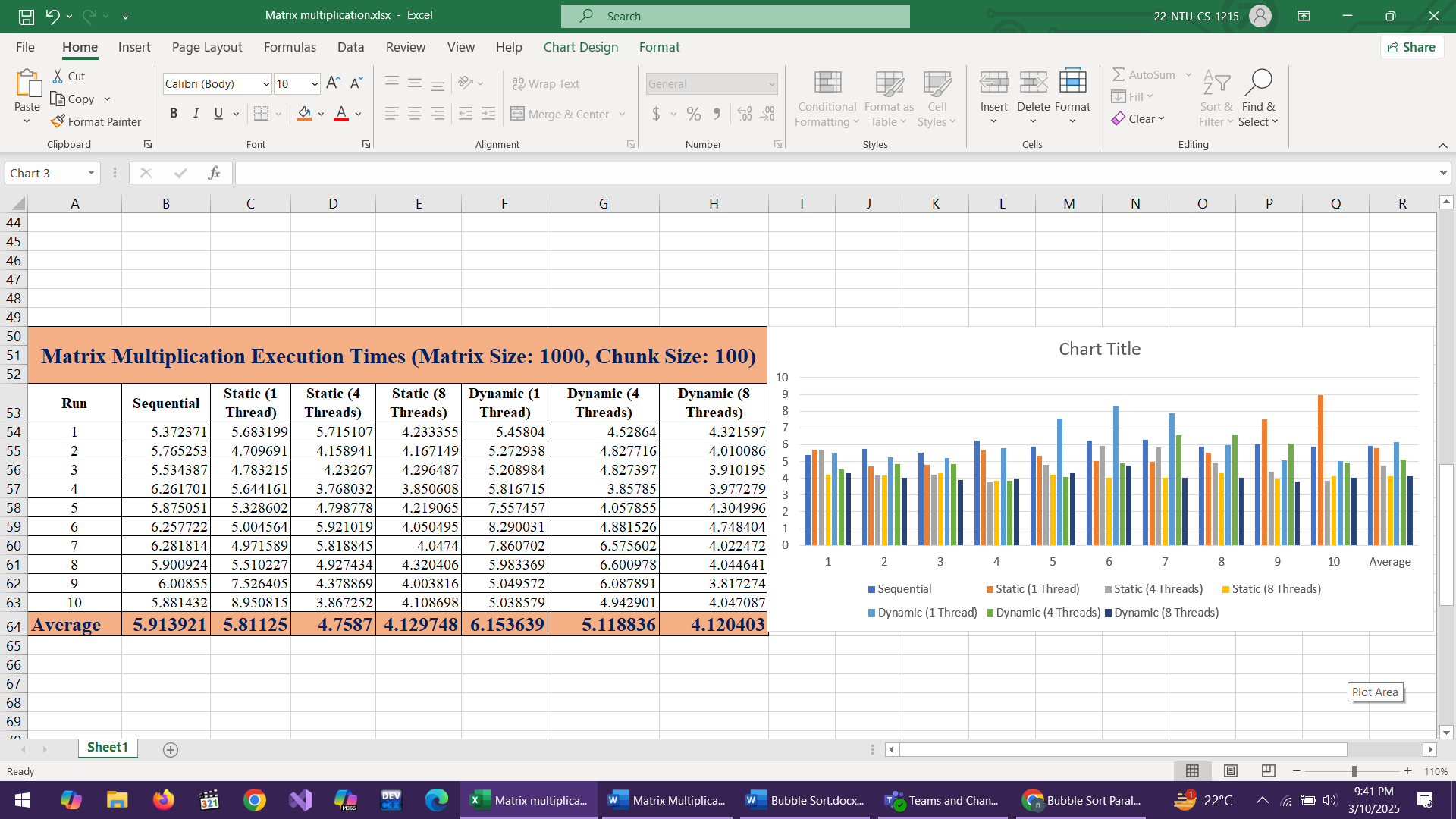
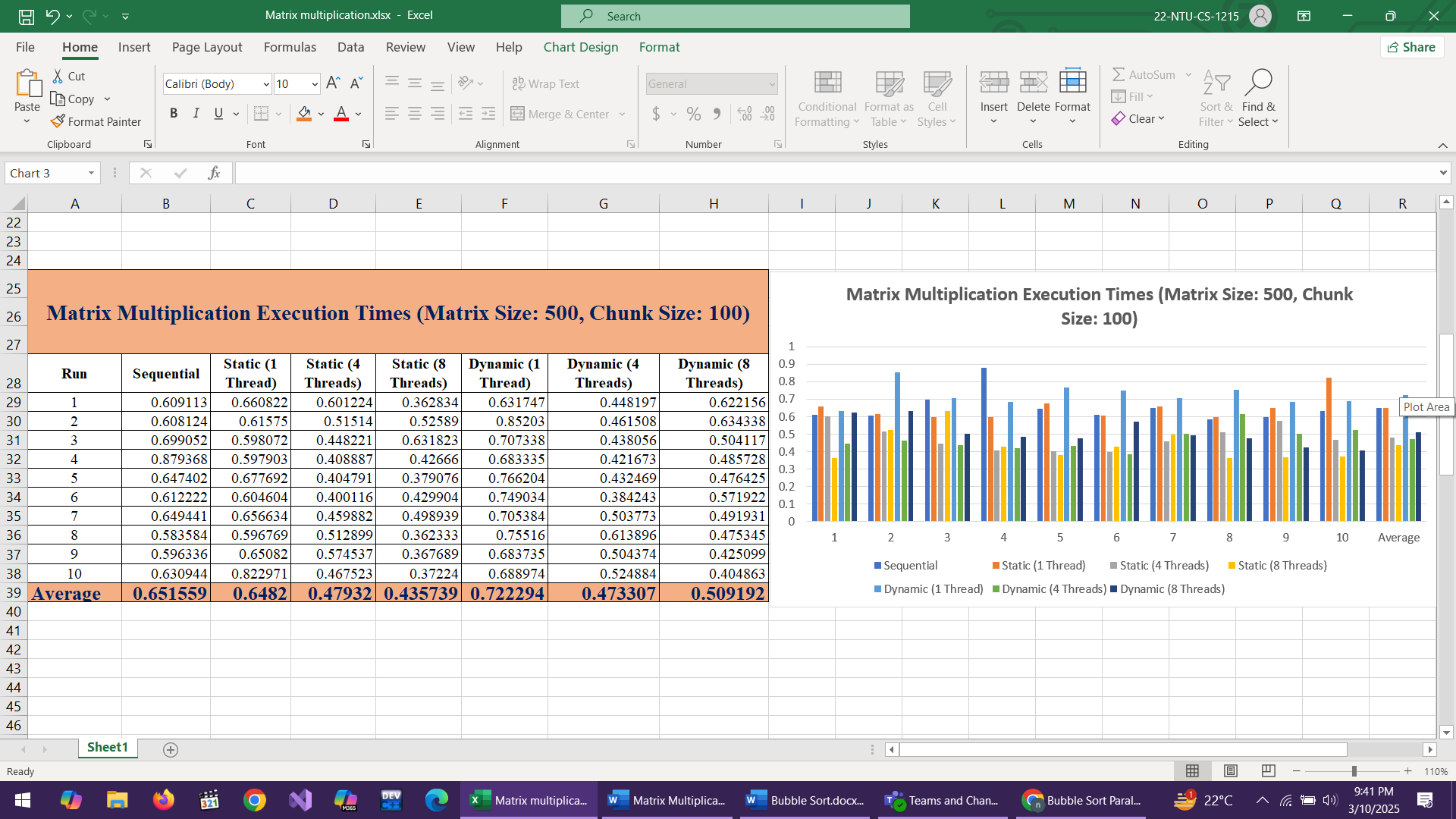
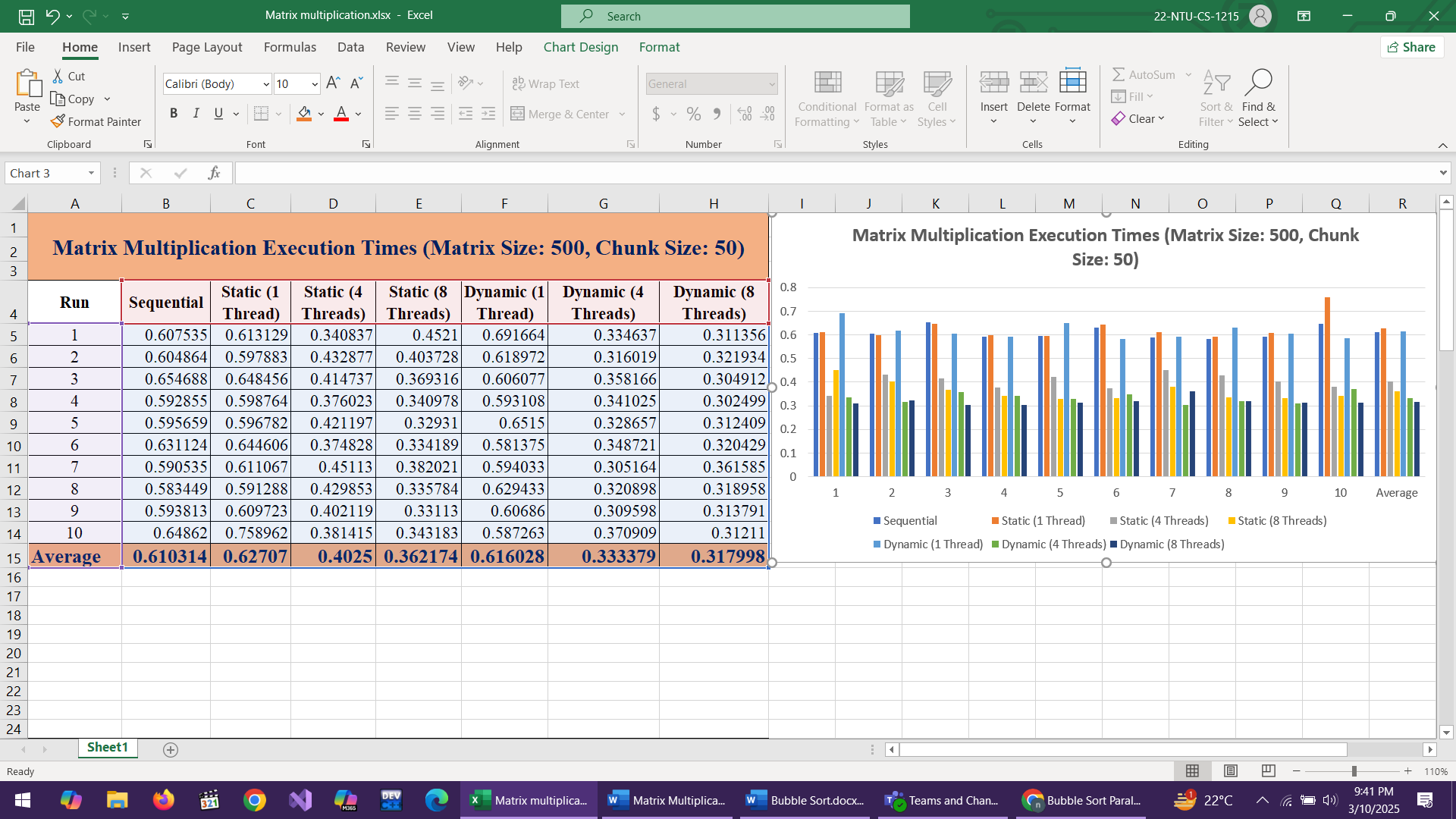
        printf("Average Time: %.6f sec\n", totalTime / 10);

    }

    return 0;

}

Dynamic Scheduling (OpenMP):  
This method also uses multiple threads, but instead of giving them fixed portions, it assigns new work dynamically as soon as a thread finishes its previous task. This ensures that no thread remains idle, making better use of all available processing power. It is especially helpful for large matrices where different parts of the matrix might take varying amounts of time to compute. Since the work is balanced efficiently, this method is generally better for large computations.



## Challenges We Faced and How We Fixed Them:

* **Stack Overflow Problem** – At first, we created the matrices inside the main function, which worked fine for small sizes. But when we used large matrices like **1000x1000**, the program crashed because of **stack overflow**. This happened because local variables are stored in a small memory area called the **stack**, which couldn’t handle such a big matrix. To fix this, we **declared the matrices globally**, which gave them more memory and stopped the crashes. Now, the program runs smoothly even for large datasets.
* **Slow Speed in Sequential Code** – The first version of the code used only **one thread**, meaning the whole calculation was done step by step, which took too long. To speed it up, we used **OpenMP** to run multiple threads at the same time. This made the program much faster. However, using too many threads also caused problems, so we had to find the right balance.
* **Static Scheduling Didn’t Always Work Well** – In **static scheduling**, each thread gets an equal part of the matrix to work on. This worked fine when all parts took the same time, but sometimes, some parts were harder to calculate, so some threads finished early and had to wait. This made performance worse.
* **Dynamic Scheduling Worked Better for Large Matrices** – To fix the waiting problem, we used **dynamic scheduling**, which gives new work to threads as soon as they finish their previous task. This made sure no thread was sitting idle. It was especially useful for **bigger matrices**, where different parts take different amounts of time to compute.
* **Choosing the Right Thread Count and Chunk Size** – Using too **few threads** made the program slow, and using too **many threads** caused extra processing overhead. We found that for **small matrices, 4 threads with static scheduling** worked best, and for **large matrices, 8 threads with dynamic scheduling** gave the best performance. Also, setting the right chunk size was important—**too big** and some threads got stuck with too much work, **too small** and there was too much switching between tasks.

## Final Conclusion:

At first, our code was slow and crashed on large datasets due to stack overflow, which we fixed by **declaring matrices globally**. To speed up calculations, we used **OpenMP parallelization**, which significantly improved performance. After testing different methods, we found that **static scheduling with 4 threads works best for small matrices**, while **dynamic scheduling with 8 threads is the best choice for large matrices**. Now, our code runs **faster, smoother, and can handle big data efficiently.**

## Lessons Learned:

1. **Stack Overflow Issue** – At first, we placed matrices inside the main function, but for large sizes, the program crashed. We fixed it by declaring matrices globally to use more memory safely.
2. **Execution Time Matters** – Running the program multiple times and averaging the results helped us understand how different scheduling methods and thread counts affect speed.
3. **Static vs. Dynamic Scheduling** – Static scheduling is simple but inefficient if some threads finish early, leaving others still working. Dynamic scheduling balances the load better but has extra overhead.
4. **Choosing the Right Thread Count** – More threads don’t always mean faster execution. If we use too many, the overhead of managing them slows down the program.
5. **Chunk Size Affects Speed** – Large chunks reduce scheduling overhead but may cause imbalance, while smaller chunks ensure even work distribution but increase scheduling time.
6. **Parallelization Benefits Large Data** – Small matrices don’t benefit much from parallelization, but as the size increases, using multiple threads helps reduce execution time.
7. **Understanding OpenMP** – Using OpenMP required careful handling of variable scopes to avoid incorrect results or unnecessary delays.
8. **Debugging Parallel Code is Harder** – Errors like race conditions and variable scope mistakes were harder to detect than in normal sequential code.
9. **Efficient Code Writing** – Optimizing code and choosing the right parallelization techniques made a huge difference in performance.
10. **Real-World Performance Optimization** – This project gave us hands-on experience in improving computational tasks and understanding high-performance computing challenges.

## Challenges with Parallelizing Bubble Sort and Transition to Matrix Multiplication:

Initially, I worked on the entire assignment using Bubble Sort, but after testing, I realized that its **sequential version was actually faster than the parallel one**. This was because Bubble Sort is **not suitable for parallelization**, as each step depends on the previous one, making it hard for multiple threads to work independently. Even after trying **static and dynamic scheduling**, performance did not improve. **Static scheduling** caused some threads to finish early while others were still working, wasting CPU power, while **dynamic scheduling** had too much memory access and waiting, slowing things down. Adding **critical sections** for safe swapping made things even worse, as threads had to wait for each other, increasing delays. Also, as the array size grew, sorting time increased significantly, and adding more threads did not help because too much thread management caused additional delays. Due to these issues, I decided to **switch from Bubble Sort to Matrix Multiplication**, which provided much better results with parallelization.

## Bubble Sorting Algorithm

## Sequential Bubble Sort:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define SIZE 5000 // Size of the array

void bubbleSort(int arr[], int n) { // Sequential Bubble Sort Function

    for (int i = 0; i < n - 1; i++) {

        for (int j = 0; j < n - i - 1; j++) {

            if (arr[j] > arr[j + 1]) { // if value of arr[j] is greater than next element arr[j+1] it swap both values to sort

                int temp = arr[j];

                arr[j] = arr[j + 1];

                arr[j + 1] = temp;

            }

        }

    }

}

int main() {

    int arr[SIZE];

    srand(time(NULL)); // It is used to get different random numbers every time the program runs, so I can test the execution time with different data.

    double total\_time = 0; //variable to store total time for 10 runs

    for (int r = 0; r < 10; r++) {  // for loop to execute bubble sort 10 times

        for (int i = 0; i < SIZE; i++) { // This for loop generates random numbers to test code

            arr[i] = rand() % 1000; //generate numbers between 0 and 999 and store on each array index

        }

        clock\_t start = clock(); // Calculate starting time before function call

        bubbleSort(arr, SIZE); // Function call to sort array

        clock\_t end = clock(); // Calculate ending time

        double execution\_time = ((double)(end - start)) / CLOCKS\_PER\_SEC; // Find time taken for this run

        total\_time += execution\_time; // Add time of this run to total time for 10 runs

        printf("Attempt %d time: %f seconds\n", r + 1, execution\_time); // Show time for each attempt

    }

    double average\_time = total\_time / 10; // Calculate average time of 10 runs

    printf("Average sequential bubble sort time over 10 attempts: %f seconds\n", average\_time); // Display average time

    return 0;

}

## Parallel Bubble Sort (static):

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define SIZE 5000  // Size of the array

// Function to perform parallel bubble sort using static scheduling

void bubbleSortParallelStatic(int arr[], int n, int numThreads) {

    for (int i = 0; i < n - 1; i++) {

        // Parallelizing the loop with OpenMP using static scheduling

        #pragma omp parallel for num\_threads(numThreads) schedule(static, 50)

        for (int j = 0; j < n - i - 1; j++) {

            if (arr[j] > arr[j + 1]) { // If the current element is greater than the next one, swap them

                #pragma omp critical  // Ensures only one thread swaps at a time

                {

                    int temp = arr[j];

                    arr[j] = arr[j + 1];

                    arr[j + 1] = temp;

                }

            }

        }

    }

}

int main() {

    int threadsList[] = {1, 4, 8};  // Different thread counts to test

    // Run sorting with different thread counts

    for (int t = 0; t < 3; t++) {

        int numThreads = threadsList[t];

        printf("\nRunning with %d threads (Static Scheduling)\n", numThreads);

        double totalTime = 0;  // Variable to store total execution time for 10 runs

        // Loop to execute 10 times

        for (int run = 0; run < 10; run++) {

            int arr[SIZE];

            // Generate random numbers for the array

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

                arr[i] = rand() % 1000;  // Fill array with numbers between 0 and 999

            }

            double startTime = omp\_get\_wtime();  // Start time measurement

            bubbleSortParallelStatic(arr, SIZE, numThreads);  // Call sorting function

            double endTime = omp\_get\_wtime();  // End time measurement

            double runTime = endTime - startTime;  // Calculate time taken for this run

            totalTime += runTime;  // Add time to total

            // Print time taken for this run

            printf("Run %d -> Threads: %d  Time: %f sec\n", run + 1, numThreads, runTime);

        }

        // Calculate and print the average execution time over 10 runs

        double avgTime = totalTime / 10;

        printf("Average time with %d threads: %f sec\n", numThreads, avgTime);

    }

    return 0;

}

## Parallel Bubble Sort (dynamic scheduling):

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define SIZE 5000  // Size of the array

// Function to perform parallel bubble sort using dynamic scheduling

void bubbleSortParallelDynamic(int arr[], int n, int numThreads) {

    for (int i = 0; i < n - 1; i++) {

        // Parallelizing the loop with OpenMP using dynamic scheduling

        #pragma omp parallel for num\_threads(numThreads) schedule(dynamic, 50)

        for (int j = 0; j < n - i - 1; j++) {

            if (arr[j] > arr[j + 1]) {  // If the current element is greater than the next one, swap them

                #pragma omp critical  // Ensures only one thread swaps at a time

                {

                    int temp = arr[j];

                    arr[j] = arr[j + 1];

                    arr[j + 1] = temp;

                }

            }

        }

    }

}

int main() {

    int threadsList[] = {1, 4, 8};  // Different thread counts to test

    // Run sorting with different thread counts

    for (int t = 0; t < 3; t++) {

        int numThreads = threadsList[t];

        printf("\nRunning with %d threads (Dynamic Scheduling)\n", numThreads);

        double totalTime = 0;  // Variable to store total execution time for 10 runs

        // Loop to execute 10 times

        for (int run = 0; run < 10; run++) {

            int arr[SIZE];

            // Generate random numbers for the array

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

                arr[i] = rand() % 1000;  // Fill array with numbers between 0 and 999

            }

            double startTime = omp\_get\_wtime();  // Start time measurement

            bubbleSortParallelDynamic(arr, SIZE, numThreads);  // Call sorting function

            double endTime = omp\_get\_wtime();  // End time measurement

            double runTime = endTime - startTime;  // Calculate time taken for this run

            totalTime += runTime;  // Add time to total

            // Print time taken for this run

            printf("Run %d -> Threads: %d  Time: %f sec\n", run + 1, numThreads, runTime);

        }

        // Calculate and print the average execution time over 10 runs

        double avgTime = totalTime / 10;

        printf("Average time with %d threads: %f sec\n", numThreads, avgTime);

    }

    return 0;

}

