CSC453: Group project
An Experimental Study of a Parallel Vectors Multiplication Algorithm

School of Computer Science, College of Computer and Information Sciences, KSU
Riyadh, Saudi Arabia

Abstract. This project examines how parallel computing can speed up the calculation of Euclidean distance between vectors, a common operation in data analysis and machine learning. We tested three methods: a simple sequential approach, OpenMP for shared-memory systems, and MPI for distributed systems. OpenMP performed best, offering faster results with less overhead, while MPI struggled with scalability due to communication delays between processors. Overall, parallelization showed clear benefits over the sequential method, highlighting the value of choosing the right approach based on the hardware. Future efforts will explore combining methods and optimizing for specific systems.

1 Introduction

The computation of Euclidean distance between data points is a fundamental operation in computer science, particularly important in fields like machine learning and data analysis. This metric quantifies the distance between two points in multi-dimensional space and is essential for various algorithms, including k-nearest neighbors and clustering methods. As datasets grow in size and complexity, the efficiency of these computations becomes increasingly critical [1]. To address this challenge, parallel processing enables simultaneous execution of tasks, significantly reducing computation time. This project explores the implementation of a parallel algorithm for calculating the Euclidean distance between two vectors through three approaches: a sequential implementation, an OpenMP parallelization, and a distributed MPI (Message Passing Interface) version. The goal is to evaluate the performance of these implementations by focusing on execution times and scalability as vector size increases [2][3]. By comparing these methods, we aim to gain insights into the effectiveness of parallel processing techniques in handling large-scale distance computations.

2 Problem definition

The Euclidean distance provides a measure of the straight-line distance between two points in a multi-dimensional space.
For two vectors a and b of equal length n, the Euclidean distance d is mathematically defined as:
$d = \sqrt{(\Sigma (a\_i - b\_i)^2)}$ for i = 1 to n
Where a_i and b_i are the components of vectors a and b, respectively.
The computation of the Euclidean distance involves two main steps: first, calculating the difference between each corresponding pair of components from the two vectors, squaring these differences, and finally summing all the squared values before taking the square root of the result. This process can be easily implemented in a sequential manner, but as the size of the vectors increases, the computational load becomes significant.
For small values of n, the sequential calculation of the Euclidean distance is straightforward and efficient. However, as n increases, the time complexity of this operation becomes a concern. The sequential algorithm typically has a time complexity of O(n). For large datasets, this can lead to slow execution times, especially in applications that require real-time processing or analysis.

3 Sequential Algorithm

3.1 Pseudo code

Algorithm 1 Sequential

```
FUNCTION Euclidean_Distance(a, b, n)
    DECLARE sum AS DOUBLE = 0.0
    FOR i FROM 0 TO n - 1
        DECLARE diff AS DOUBLE = a[i] - b[i]
        sum = sum + (diff * diff) // Sum of squares of differences
    END FOR
    RETURN sqrt(sum) // Return the square root of the sum
END FUNCTION
FUNCTION main()
    DECLARE n AS INTEGER
    PRINT "Enter vector length: "
    READ n // Get vector length from user input
    IF n <= 0 THEN
        PRINT "Vector length must be a positive integer."
        RETURN 1
    END IF
    // Seed random number generator
    CALL srand(time(NULL))
    // Allocate memory for two vectors
    DECLARE a AS ARRAY OF DOUBLE WITH SIZE n
    DECLARE b AS ARRAY OF DOUBLE WITH SIZE n
    IF a IS NULL OR b IS NULL THEN
        PRINT "Memory allocation failed."
        RETURN 1
    END IF
    DECLARE total_time AS DOUBLE = 0.0 // Variable to hold total time
    DECLARE runs AS INTEGER = 10 // Number of runs
    FOR r FROM 0 TO runs - 1
        // Generate random values for vectors
        FOR i FROM 0 TO n - 1
            a[i] = RANDOM_VALUE_BETWEEN(0 AND 99) // Random values between 0 and 99
            b[i] = RANDOM_VALUE_BETWEEN(0 AND 99)
        END FOR
        // Start timing
        DECLARE start_time AS CLOCK_T = clock()
```

```
        // Calculate the Euclidean distance
        DECLARE distance AS DOUBLE = Euclideandistance(a, b, n)
        // End timing
        DECLARE end_time AS CLOCK_T = clock()
        // Calculate elapsed time in seconds
        DECLARE elapsed_time AS DOUBLE = (end_time - start_time) / CLOCKS_PER_SEC
        total_time = total_time + elapsed_time // Accumulate total time
        // Print the result for this run
        PRINT "Run " + (r + 1) + ": Euclidean distance = " + distance + ", Time taken = " + elapsed_time + " seconds"
    END FOR
    // Calculate and print average time
    DECLARE average_time AS DOUBLE = total_time / runs
    PRINT "Average time taken over " + runs + " runs: " + average_time + " seconds"
    // Free allocated memory
    FREE(a)
    FREE(b)
    RETURN 0
END FUNCTION
```

3.2 Performance

The time complexity of the Euclidean distance calculation algorithm is \(O(n)\), where \(n\) is the size of the vectors. In my tests, I used vectors of size 1,000,000. I ran the sequential code on my PC, which has 16 logical processors, measuring the execution time over 10 runs using the `clock()` function for accuracy. The average time taken for the calculations was approximately 0.003808 seconds. This result demonstrates the efficiency of the algorithm even when processing large vectors, highlighting its capability to perform effectively on systems with multiple logical processors.

4 OpenMP Algorithm

4.1 OpenMP pseudo code

Algorithm 2 OpenMP

```
N = 10000                // Size of the vectors
NUM_TRIALS = 10          // Number of trials to average

function rand_float():
   return random() / MAX_RANDOM_VALUE

function main():
      disable_stdout_buffering()
   print("Vector size:", N)
   num_procs = get_number_of_processors()  // Get number of processors
   print("Number of processors:", num_procs)

   vector1 = allocate_memory(N)
   vector2 = allocate_memory(N)

     seed_random_generator(42)
   for i = 0 to N-1:
      vector1[i] = rand_float()
      vector2[i] = rand_float()

      print("First 5 values of vector1:", vector1[0:5])
   print("First 5 values of vector2:", vector2[0:5])

      seq_distance = 0
   for i = 0 to N-1:
      diff = vector1[i] - vector2[i]
      seq_distance += diff * diff
   seq_distance = sqrt(seq_distance)
   print("Sequential Euclidean Distance (for verification):", seq_distance)

   for threads in [2, 4, 8, 16]:
      set_num_threads(threads)  // Set number of OpenMP threads
      print("Running with", threads, "threads:")
      total_time = 0

      for trial = 1 to NUM_TRIALS:
         start_time = get_current_time()

         euclidean_distance = 0
         parallel for i = 0 to N-1 with reduction(+):
            diff = vector1[i] - vector2[i]
            euclidean_distance += diff * diff
         euclidean_distance = sqrt(euclidean_distance)

         end_time = get_current_time()
         time_elapsed = end_time - start_time
         total_time += time_elapsed
         print("Trial", trial, "- Euclidean Distance:", euclidean_distance,
               ", Computation Time:", time_elapsed, "seconds")

      average_time = total_time / NUM_TRIALS
      print("Average Computation Time with", threads, "threads:", average_time, "seconds")

   free_memory(vector1)
   free_memory(vector2)
```

4.2 OpenMP performance

In the OpenMP implementation of this Euclidean distance calculation, the work is divided among threads by splitting the vector indices, so each thread calculates the squared difference for its assigned elements in parallel, with results accumulated using a reduction clause to sum the distances. The algorithm's complexity is O(N)O(N)O(N), as the work required grows linearly with the vector size NNN and does not decrease with additional threads, though parallelism reduces computation time by dividing the workload. Using omp_get_wtime() to measure computation time, the code was run with a vector size of 10,000 on a machine with a specified number of processors. We evaluated performance with 2, 4, 8, and 16 threads, averaging times over 10 trials per thread configuration to account for variability. The average computation times were as follows: 0.003388 seconds for 2 threads, 0.002979 seconds for 4 threads, 0.003444 seconds for 8 threads, and 0.004489 seconds for 16 threads. Initially, increasing the thread count improved performance, but diminishing returns appeared as overhead and system limits became more significant at higher thread counts.

5 MPI Algorithm

5.1 MPI pseudo code

Algorithm 3 MPI

```
// Declare necessary variables
my_id ← rank of the processor
num_process ← total number of processors
n ← vector length
total_time ← 0
a, b ← vectors of size n
local_sum ← local sum of squared differences
total_sum ← total sum of squared differences
local_start, local_end ← local indices for each processor

// Initialize MPI environment
MPI_Init()
my_id ← MPI_Comm_rank(MPI_COMM_WORLD)
num_process ← MPI_Comm_size(MPI_COMM_WORLD)

if (argc != 2) then // Parse the vector length (n) from the command line argument
    if (my_id == 0) then
        print "Usage: <program_name> <vector_length>"
    end if
    MPI_Abort(MPI_COMM_WORLD, 1)
end if
n ← atoi(argv[1])  // vector length specified by the user

a, b ← allocate memory for vectors of size n

if (my_id == 0) then // Root process generates random vectors (a and b)
    seed ← current time in seconds
    for i ← 0 to n-1 do
        a[i] ← random value between 0 and 1
        b[i] ← random value between 0 and 1
    end for
end if

MPI_Bcast(n, 1, MPI_INT, 0, MPI_COMM_WORLD) // Broadcast vector length and vectors to all processes
MPI_Bcast(a, n, MPI_DOUBLE, 0, MPI_COMM_WORLD)
MPI_Bcast(b, n, MPI_DOUBLE, 0, MPI_COMM_WORLD)

for run ← 1 to 10 do
    MPI_Barrier(MPI_COMM_WORLD)  // Synchronize all processes before starting the timer
    start_time ← MPI_Wtime()  // Start measuring time for this run

    chunk_size ← n / num_process // Calculate the local range for each process
    local_start ← my_id * chunk_size
    local_end ← (my_id == num_process - 1) ? n : local_start + chunk_size

    local_sum ← 0
    for i ← local_start to local_end - 1 do
        local_sum ← local_sum + (a[i] - b[i])^2
    end for

    MPI_Reduce(local_sum, total_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD) // Reduce all local sums to process 0
using MPI_Reduce

    MPI_Barrier(MPI_COMM_WORLD) // Synchronize all processes after computation

    if (my_id == 0) then
        distance ← sqrt(total_sum)
        print "Run ", run, ": Euclidean Distance: ", distance
        end_time ← MPI_Wtime() // Calculate and print the time taken for this run
        time_taken ← end_time - start_time
        print "Run ", run, ": Time taken: ", time_taken
        total_time ← total_time + time_taken  // Accumulate the total time
    end if
end for

if (my_id == 0) do // Calculate and print the average time after all runs
    average_time ← total_time / num_runs
    print "Average Time taken: ", average_time
```

```
end if

free(a) , free(b) // Free memory after all runs
MPI_Finalize() // Finalize MPI environment
```

5.2 MPI performance

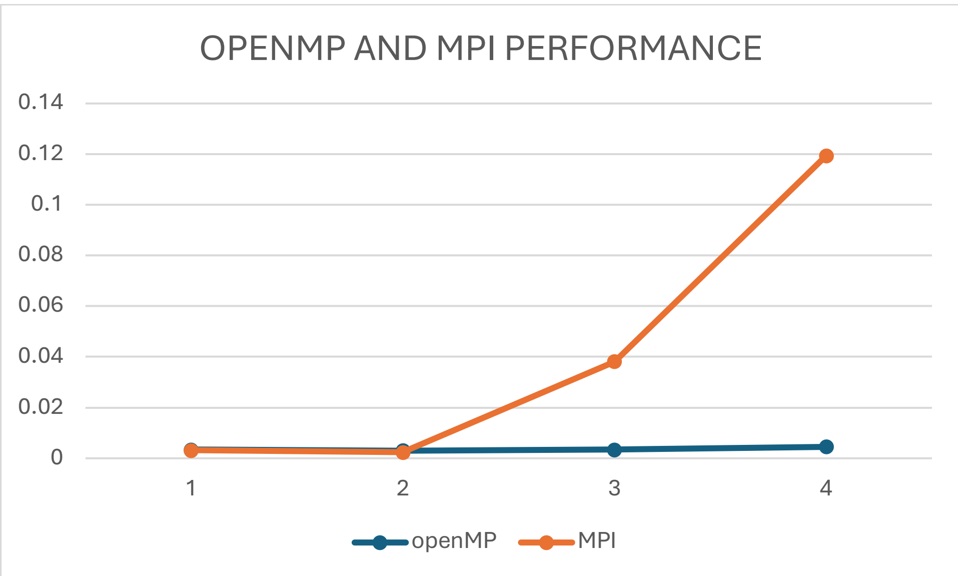
In this parallel Euclidean distance computation using MPI, the workload is distributed across multiple processors by dividing the vector into equal parts, with each processor handling a specific segment of the data. The algorithm has a time complexity of O(n), where n is the number of elements in the vector. The work was done on a MacBook Pro with 4 processors. For a vector of size 1,00,000 , the average times measured over 10 runs are as follows: The average time for 2 processors is 0.003059 seconds, for 4 processors is 0.002409 seconds, for 8 processors is 0.038194 seconds, and finally, for 16 processors, the average time is 0.119361 seconds. As you can see, there was a slight improvement in performance between 2 and 4 processors. However, the time increased significantly with 8 and 16 processors, which can be attributed to the communication overhead of MPI. As more processes are used, the need for synchronization and data exchange between processes grows, leading to higher latency and less efficiency in scaling.

6 Result and discussion

Based on the test results, **Table 1** summarizes the performance of each method. Parallelization improves efficiency compared to the sequential approach. However, the performance of the parallelization algorithms varies depending on the context; OpenMP may be more efficient under certain conditions, while MPI could outperform it in others. The graph below highlights the relative efficiency of the two parallelization methods, showcasing their advantages and limitations in different scenarios.

Number of Threads /Processors	***1***	**2**	**4**	**8**	**16**
Sequential	0.003808				
OpenMP		0.003388	0.002979	0.003444	0.004489
MPI		0.003092	0.002409	0.038197	0.119361

Table 1 Summarize of sequential, OpenMP, and MPI algorithm's performance



7 Conclusion and future work

To improve the efficiency of vector multiplication and accelerate computation, this project evaluated two parallelization methods, OpenMP and MPI, alongside a sequential baseline. The parallel implementations demonstrated significant performance improvements compared to the sequential approach. Among the parallel methods, OpenMP consistently outperformed MPI, particularly in shared-memory environments, due to its efficient workload distribution and minimal communication overhead. While MPI showed its strengths in distributed systems, its performance was limited by inter-process communication costs. These findings highlight the advantages of parallelization and emphasize the importance of selecting the appropriate method based on the target hardware and computational requirements.

References

1: Bishop, C. M. (2006). Pattern Recognition and Machine Learning. Springer.

2: Gropp, W., Lusk, E., & Thakur, R. (2014). Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press.

3: Dagum, L., & Menon, R. (1998). OpenMP: An Industry Standard API for Shared-Memory Programming. IEEE Computational Science and Engineering, 5(1), 46-55.

4: LNCS Homepage, http://www.springer.com/lncs, last accessed 2016/11/21.

Appendix A

Sequential run results

Output Clear

```
Enter vector length: 1000000
Run 1: Euclidean distance = 40827.410988, Time taken = 0.003800
    seconds
Run 2: Euclidean distance = 40829.372589, Time taken = 0.003840
    seconds
Run 3: Euclidean distance = 40813.349838, Time taken = 0.003830
    seconds
Run 4: Euclidean distance = 40842.132315, Time taken = 0.003810
    seconds
Run 5: Euclidean distance = 40803.728690, Time taken = 0.003783
    seconds
Run 6: Euclidean distance = 40802.350422, Time taken = 0.003803
    seconds
Run 7: Euclidean distance = 40837.434310, Time taken = 0.003777
    seconds
Run 8: Euclidean distance = 40825.853794, Time taken = 0.003766
    seconds
Run 9: Euclidean distance = 40840.202081, Time taken = 0.003890
    seconds
Run 10: Euclidean distance = 40799.087784, Time taken = 0.003780
    seconds
Average time taken over 10 runs: 0.003808 seconds


=== Code Execution Successful ===
```

OpenMP run results

```
Vector size: 1000000
Number of processors: 4
First 5 values of vector1: 0.0003 0.7354 0.3762 0.9759 0.5304
First 5 values of vector2: 0.5246 0.2633 0.1963 0.5123 0.2571
Sequential Euclidean Distance (for verification): 408.05069731
Running with 2 threads:
Trial 1 - Euclidean Distance: 408.05069731, Computation Time: 0.003859 seconds
Trial 2 - Euclidean Distance: 408.05069731, Computation Time: 0.004001 seconds
Trial 3 - Euclidean Distance: 408.05069731, Computation Time: 0.004890 seconds
Trial 4 - Euclidean Distance: 408.05069731, Computation Time: 0.003920 seconds
Trial 5 - Euclidean Distance: 408.05069731, Computation Time: 0.002742 seconds
Trial 6 - Euclidean Distance: 408.05069731, Computation Time: 0.002760 seconds
Trial 7 - Euclidean Distance: 408.05069731, Computation Time: 0.002272 seconds
Trial 8 - Euclidean Distance: 408.05069731, Computation Time: 0.002340 seconds
Trial 9 - Euclidean Distance: 408.05069731, Computation Time: 0.003189 seconds
Trial 10 - Euclidean Distance: 408.05069731, Computation Time: 0.003905 seconds
Average Computation Time with 2 threads: 0.003388 seconds

Running with 4 threads:
Trial 1 - Euclidean Distance: 408.05069731, Computation Time: 0.002461 seconds
Trial 2 - Euclidean Distance: 408.05069731, Computation Time: 0.002244 seconds
Trial 3 - Euclidean Distance: 408.05069731, Computation Time: 0.001810 seconds
Trial 4 - Euclidean Distance: 408.05069731, Computation Time: 0.002061 seconds
Trial 5 - Euclidean Distance: 408.05069731, Computation Time: 0.003194 seconds
Trial 6 - Euclidean Distance: 408.05069731, Computation Time: 0.003963 seconds
Trial 7 - Euclidean Distance: 408.05069731, Computation Time: 0.003935 seconds
Trial 8 - Euclidean Distance: 408.05069731, Computation Time: 0.003526 seconds
Trial 9 - Euclidean Distance: 408.05069731, Computation Time: 0.003063 seconds
Trial 10 - Euclidean Distance: 408.05069731, Computation Time: 0.003535 seconds
Average Computation Time with 4 threads: 0.002979 seconds

Running with 8 threads:
Trial 1 - Euclidean Distance: 408.05069731, Computation Time: 0.003190 seconds
Trial 2 - Euclidean Distance: 408.05069731, Computation Time: 0.002831 seconds
Trial 3 - Euclidean Distance: 408.05069731, Computation Time: 0.002757 seconds
Trial 4 - Euclidean Distance: 408.05069731, Computation Time: 0.002664 seconds
Trial 5 - Euclidean Distance: 408.05069731, Computation Time: 0.003022 seconds
Trial 6 - Euclidean Distance: 408.05069731, Computation Time: 0.004319 seconds
Trial 7 - Euclidean Distance: 408.05069731, Computation Time: 0.003602 seconds
Trial 8 - Euclidean Distance: 408.05069731, Computation Time: 0.004402 seconds
Trial 9 - Euclidean Distance: 408.05069731, Computation Time: 0.003482 seconds
Trial 10 - Euclidean Distance: 408.05069731, Computation Time: 0.004167 seconds
Average Computation Time with 8 threads: 0.003444 seconds

Running with 16 threads:
Trial 1 - Euclidean Distance: 408.05069731, Computation Time: 0.004673 seconds
Trial 2 - Euclidean Distance: 408.05069731, Computation Time: 0.005657 seconds
Trial 3 - Euclidean Distance: 408.05069731, Computation Time: 0.006161 seconds
Trial 4 - Euclidean Distance: 408.05069731, Computation Time: 0.003366 seconds
Trial 5 - Euclidean Distance: 408.05069731, Computation Time: 0.003681 seconds
Trial 6 - Euclidean Distance: 408.05069731, Computation Time: 0.002995 seconds
Trial 7 - Euclidean Distance: 408.05069731, Computation Time: 0.006311 seconds
Trial 8 - Euclidean Distance: 408.05069731, Computation Time: 0.003527 seconds
Trial 9 - Euclidean Distance: 408.05069731, Computation Time: 0.003524 seconds
Trial 10 - Euclidean Distance: 408.05069731, Computation Time: 0.004998 seconds
Average Computation Time with 16 threads: 0.004489 seconds
```

MPI run results

```
(base) alyaaljarallah@Alyas-MacBook-Pro Desktop % mpicc -o p1 p1.c
(base) alyaaljarallah@Alyas-MacBook-Pro Desktop % mpirun -np 2 ./p1 1000000
Run 1: Euclidean Distance: 408.158866
Run 1: Time taken: 0.006221 seconds
Run 2: Euclidean Distance: 577.223804
Run 2: Time taken: 0.002562 seconds
Run 3: Euclidean Distance: 706.951893
Run 3: Time taken: 0.003535 seconds
Run 4: Euclidean Distance: 816.317732
Run 4: Time taken: 0.002333 seconds
Run 5: Euclidean Distance: 912.670970
Run 5: Time taken: 0.002988 seconds
Run 6: Euclidean Distance: 999.780956
Run 6: Time taken: 0.002309 seconds
Run 7: Euclidean Distance: 1079.886855
Run 7: Time taken: 0.002354 seconds
Run 8: Euclidean Distance: 1154.447608
Run 8: Time taken: 0.002628 seconds
Run 9: Euclidean Distance: 1224.476598
Run 9: Time taken: 0.002563 seconds
Run 10: Euclidean Distance: 1290.711664
Run 10: Time taken: 0.003092 seconds
Average Time taken: 0.003059 seconds
```

```
(base) alyaaljarallah@Alyas-MacBook-Pro Desktop % mpirun -np 4 ./p1 1000000
Run 1: Euclidean Distance: 407.938937
Run 1: Time taken: 0.004265 seconds
Run 2: Euclidean Distance: 576.912778
Run 2: Time taken: 0.002226 seconds
Run 3: Euclidean Distance: 706.570966
Run 3: Time taken: 0.001734 seconds
Run 4: Euclidean Distance: 815.877874
Run 4: Time taken: 0.001656 seconds
Run 5: Euclidean Distance: 912.179194
Run 5: Time taken: 0.001538 seconds
Run 6: Euclidean Distance: 999.242242
Run 6: Time taken: 0.003726 seconds
Run 7: Euclidean Distance: 1079.304978
Run 7: Time taken: 0.004442 seconds
Run 8: Euclidean Distance: 1153.825555
Run 8: Time taken: 0.001454 seconds
Run 9: Euclidean Distance: 1223.816811
Run 9: Time taken: 0.001521 seconds
Run 10: Euclidean Distance: 1290.016188
Run 10: Time taken: 0.001525 seconds
Average Time taken: 0.002409 seconds
```

```
(base) alyaaljarallah@Alyas-MacBook-Pro Desktop % mpirun -np 8 ./p1 1000000
Run 1: Euclidean Distance: 408.459456
Run 1: Time taken: 0.126547 seconds
Run 2: Euclidean Distance: 577.648902
Run 2: Time taken: 0.065688 seconds
Run 3: Euclidean Distance: 707.472531
Run 3: Time taken: 0.023658 seconds
Run 4: Euclidean Distance: 816.918912
Run 4: Time taken: 0.033347 seconds
Run 5: Euclidean Distance: 913.343110
Run 5: Time taken: 0.019531 seconds
Run 6: Euclidean Distance: 1000.517248
Run 6: Time taken: 0.023796 seconds
Run 7: Euclidean Distance: 1080.682141
Run 7: Time taken: 0.030258 seconds
Run 8: Euclidean Distance: 1155.297805
Run 8: Time taken: 0.020081 seconds
Run 9: Euclidean Distance: 1225.378368
Run 9: Time taken: 0.018481 seconds
Run 10: Euclidean Distance: 1291.662213
Run 10: Time taken: 0.020586 seconds
Average Time taken: 0.038197 seconds
```

```
(base) alyaaljarallah@Alyas-MacBook-Pro Desktop % mpirun -np 16 ./p1 1000000
Run 1: Euclidean Distance: 408.183743
Run 1: Time taken: 0.541158 seconds
Run 2: Euclidean Distance: 577.258985
Run 2: Time taken: 0.196167 seconds
Run 3: Euclidean Distance: 706.994981
Run 3: Time taken: 0.079388 seconds
Run 4: Euclidean Distance: 816.367485
Run 4: Time taken: 0.084246 seconds
Run 5: Euclidean Distance: 912.726596
Run 5: Time taken: 0.080639 seconds
Run 6: Euclidean Distance: 999.841890
Run 6: Time taken: 0.043254 seconds
Run 7: Euclidean Distance: 1079.952672
Run 7: Time taken: 0.042168 seconds
Run 8: Euclidean Distance: 1154.517969
Run 8: Time taken: 0.041011 seconds
Run 9: Euclidean Distance: 1224.551228
Run 9: Time taken: 0.043512 seconds
Run 10: Euclidean Distance: 1290.790330
Run 10: Time taken: 0.042069 seconds
Average Time taken: 0.119361 seconds
```

Appendix B

TASK	STUDENT'S NAME
INTRODUCTION AND PROBLEM DEFINITION	
SEQUENTIAL PART	
OPENMP PART	
MPI	
RESULT AND DISCUSSION	
CONCLUSION	