**Ans. to Question-1**

**Question-1(a), (b):**

| **Subquestion (ϵ = 5 )** | **#Cores** | **Time (s) No Opt.** | **Time (s) w/ -O3** | **Speedup** | **Parallel Efficiency** |
| --- | --- | --- | --- | --- | --- |
| Sequential | 1 | 525.425098 |  | 1 | 1 |
| (a) | 6 | 91.547534 |  | 5.739 | 0.9565 |
| Sequential | 1 |  | 14.3133043 | 1 | 1 |
| (b) | 6 |  | 2.69447 | 5.3121 | 0.8854 |

**Table 1.1:** Measurements and metrics for sub-questions in Question 1 at **ϵ = 5**. All response times have been averaged over 3 time trials, each using separate executions of the program. The column “No Opt.” refers to executing the program without compiler optimization.

| **Subquestion (ϵ = 10 )** | **#Cores** | **Time (s) No Opt.** | **Time (s) w/ -O3** | **Speedup** | **Parallel Efficiency** |
| --- | --- | --- | --- | --- | --- |
| Sequential | 1 | 518.5767 |  | 1 | 1 |
| (a) | 6 | 92.240620 |  | 5.621999 | 0.937 |
| Sequential | 1 |  | 14.408097 | 1 | 1 |
| (b) | 6 |  | 2.731017 | 5.2757 | 0.879 |

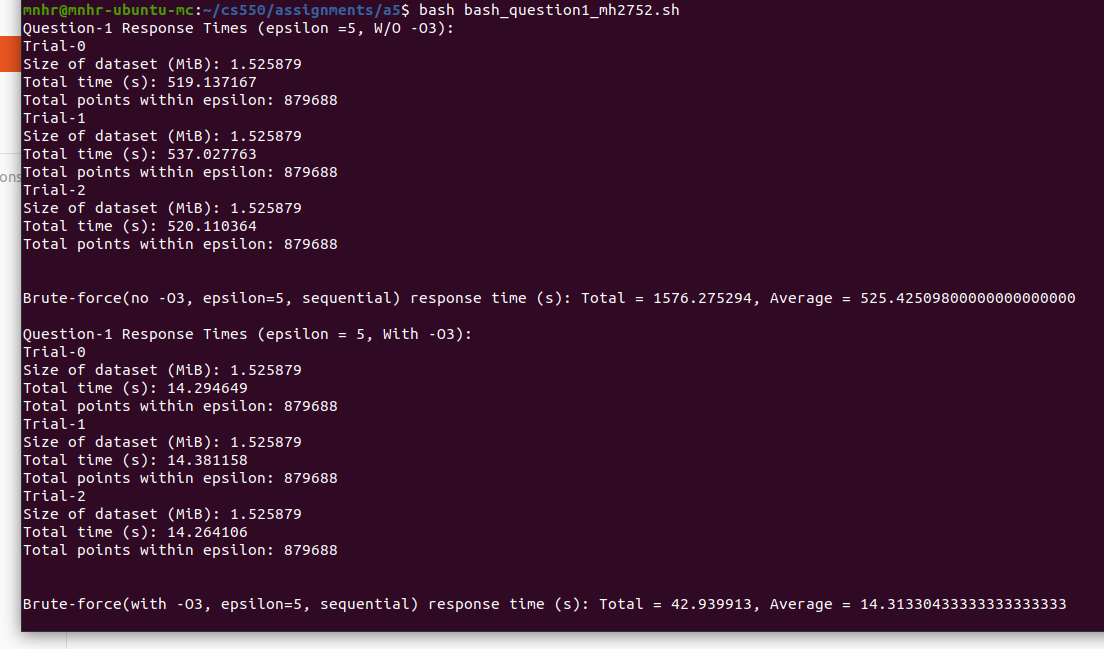
**Table 1.2:** Measurements and metrics for sub-questions in Question 1 at **ϵ = 10**. All response times have been averaged over 3 time trials, each using separate executions of the program. The column “No Opt.” refers to executing the program without compiler optimization.

**Question-1(c):**

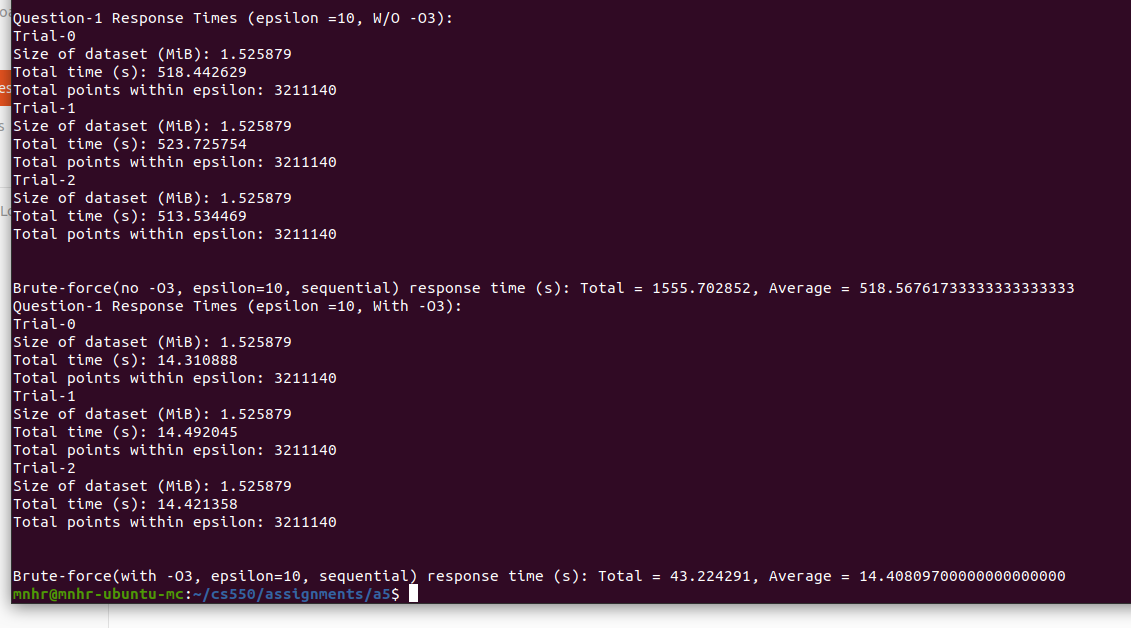
With and without the **-O3** optimization, we see the parallel version achieves **5x** speedup over the sequential version for both values of the epsilon. As the brute-force algorithm calculates the distance between a point with all the other points, parallelizing the solution helps divide the workload of **N2** distance calculations equally among the threads, hence the performance gains.

The parallel efficiencies in all four cases are considerably high (the minimum efficiency is 87.9%) as well as the speedups. This is a good thing as it tells us that a major fraction of the code-base is parallelizable which would lead to lower response times with the availability of more physical cores and threads.

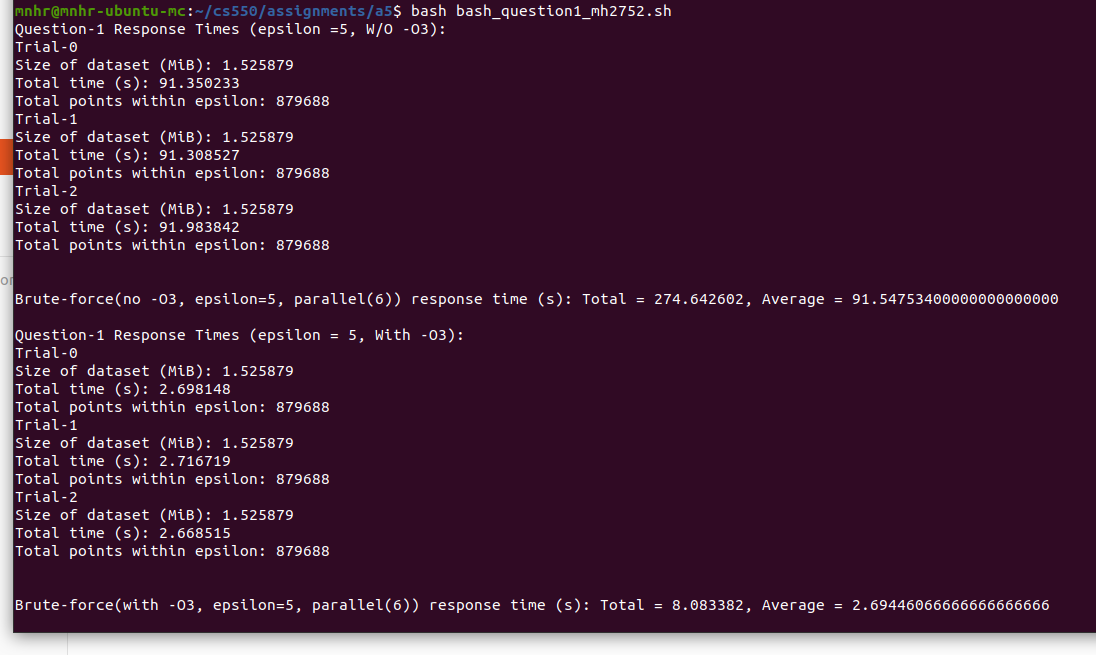
**Screenshots for Question-1**



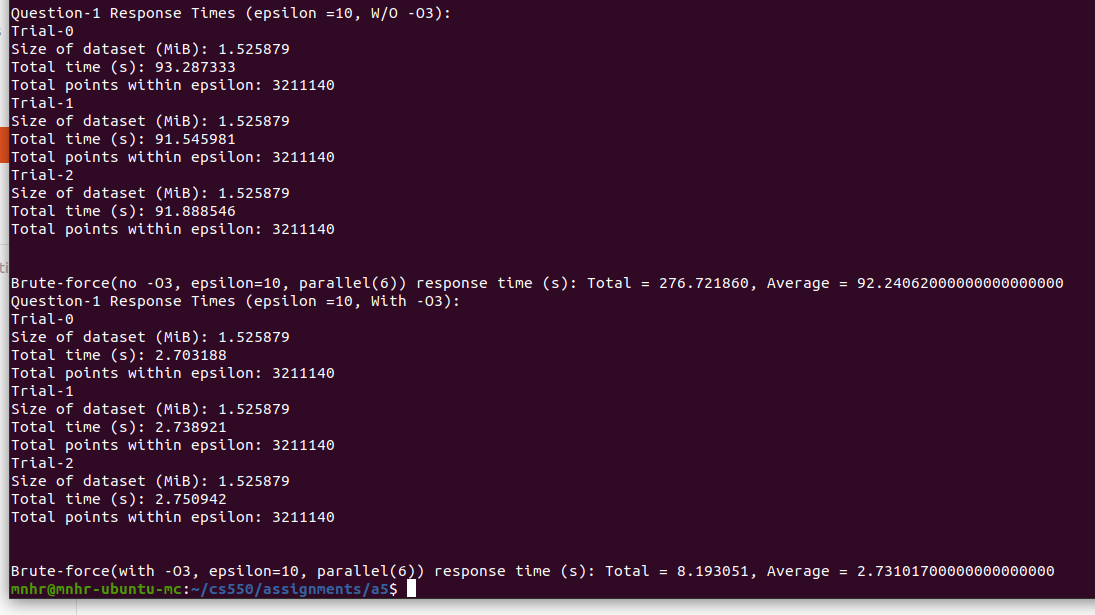
**Fig 1:** Brute-force solution (***sequential***) response times **(top)** without using -O3 and **(bottom)** using -O3 for epsilon = **5**, N = **100,000** (Average of 3 independent runs)



**Fig 2:** Brute-force solution (***sequential***) response times **(top)** without using -O3 and **(bottom)** using -O3 for epsilon = **10**, N = **100,000** (Average of 3 independent runs)



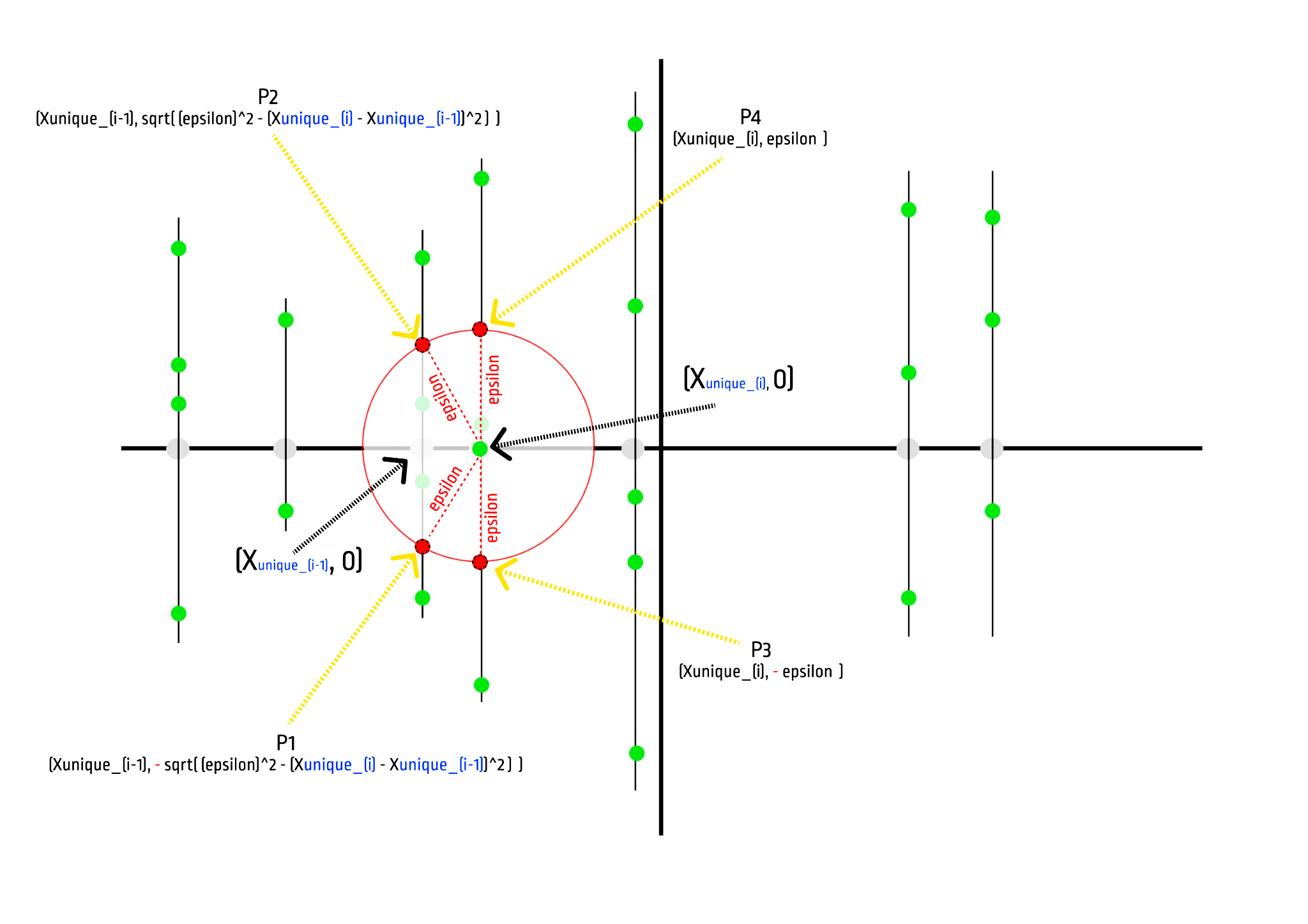
**Fig 3:** Brute-force solution (***parallel***) response times **(top)** without using -O3 and **(bottom)** using -O3 for epsilon = **5**, N = **100,000** (Average of 3 independent runs)



**Fig 4:** Brute-force solution (***parallel***) response times **(top)** without using -O3 and **(bottom)** using -O3 for epsilon = **10**, N = **100,000** (Average of 3 independent runs)

**Ans. to Question-2 (Version: 1-Nazmul(mh2752))**

**Question-2(a):**

****

**Illustration-1:** A sample execution of the optimized algorithm

The optimized algorithm reduces the number of distance calculations by approaching the given problem from a geometric point of view. It requires performing several pre-processing steps on the given **Point2D** data (**question2\_mh2752.c**) array. The optimized algorithm first sorts the given Point2D array along the x-axis. It then finds all the unique x-values from the points (the values are in sorted (increasing) order now).

For each unique x-value, a corresponding Unique\_X struct object is created and placed in an array of **Unique\_X** objects. This array of **Unique\_X** objects holds the objects in sorted order (increasing x values).

Each **Unique\_X** struct object holds the following information fields:

* **‘x\_value’**: the unique x-value associated with the Unique\_X object (a double type variable)
* **‘num\_of\_y\_values’**: total number of y-values associated with this unique ‘x\_value’ (an int type variable)
* **‘all\_y\_values’**: an array of double type values representing all the ‘y-values’ associated with this unique ‘x-value’ in sorted (increasing) order
* **‘num\_of\_left\_neighbors’**: The total number of neighboring Unique\_X objects to this ‘Unique\_X’ object’s left side within the ‘epsilon’ radius (an int type variable)
* **‘num\_of\_right\_neighbors’**: The total number of neighboring Unique\_X objects to this ‘Unique\_X’ object’s right side within the ‘epsilon’ radius (an int type variable)
* **‘left\_neighbor\_indices’**: an integer array of length ‘num\_of\_left\_neighbors’ holding the indices of the neighboring Unique\_X objects to this ‘Unique\_X’ object’s left side within the ‘epsilon’ radius from the array of Unique\_X objects
* **‘right\_neighbor\_indices’**: an integer array of length ‘num\_of\_right\_neighbors’ holding the indices of the neighboring Unique\_X objects to this ‘Unique\_X’ object’s right side within the ‘epsilon’ radius from the array of Unique\_X objects

After all the pre-processing steps are completed, the array of Unique\_X objects will represent the 2D cartesian grid populated with the given Point2D objects - something similar to Illustration-1 above.

As a result, for a single data point on the grid, the algorithm decreases the number of total distance calculations from **N** to something significantly smaller which is determined by the value of epsilon and the number of left and right Unique\_X neighbors of that point - not the total number of points **N**.

For example, let, we want to count the total number of points that fall within the **epsilon** radius of the point in **Illustration-1** indicated by the (Xunique\_(i), 0) cartesian coordinates. The optimized algorithm imagines a circle of radius **epsilon** whose center is at (Xunique\_(i), 0). Using simple geometric calculations, the algorithm then determines the cartesian coordinates of the points where the circle intersects only the neighboring vertical lines represented by neighboring Unique\_X objects. In this case, in Illustration-1, these points are P1, P2, P3, and P4. Once those coordinates are determined, the algorithm then uses a slightly modified version of the binary search algorithm to obtain the total number of points from each vertical line that falls within the range determined by the intersecting points (e.g., there are two points on the vertical line left of (Xunique\_(i), 0) point who fall within the range [P1, P2]).

As a direct consequence of using this method, the total number of distance calculations for (Xunique\_(i), 0) point is reduced from 25 **(N = 25** in Figure-1, the total number of green dots**)** to only 4. As apparent from the figure, this number will be governed by the value of epsilon and the number of neighboring Unique\_X objects **(i.e.,** how spread over/compactly placed the data points are on the 2D cartesian plane**)**.

**Question-2 (b), (c), (d):**

| **Subquestion (ϵ = 5 )** | **#Cores** | **Time (s) No Opt.** | **Time (s) w/ -O3** | **Speedup** | **Parallel Efficiency** | **Corresponding Brute-force Time (s)** | **Brute-force Vs. Optimized Algo. Ratio**  **(d)** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Sequential | 1 | 6.54665033 |  | 1 | 1 | 525.425098 | 80.2586 |
| **(b)** | 6 | 1.63103167 |  | 4.0138 | 0.669 | 91.547534 | 56.128606 |
| Sequential | 1 |  | 0.96896 | 1 | 1 | 14.3133043 | 14.772 |
| **(c)** | 6 |  | 0.399947 | 2.42 | 0.403 | 2.69447 | 6.69 |

**Table 2.1.1:** Measurements and metrics for sub-questions in Question 2 at **ϵ = 5**. All response times have been averaged over 3 time trials, each using separate executions of the program. The column “No Opt.” refers to executing the program without compiler optimization

| **Subquestion (ϵ = 10 )** | **#Cores** | **Time (s) No Opt.** | **Time (s) w/ -O3** | **Speedup** | **Parallel Efficiency** | **Corresponding Brute-force Time (s)** | **Brute-force Vs. Optimized Algo. Ratio**  **(d)** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Sequential | 1 | 13.23123 |  | 1 | 1 | 518.5767 | 39.1934 |
| **(b)** | 6 | 3.48640 |  | 3.795 | 0.633 | 92.240620 | 26.457 |
| Sequential | 1 |  | 1.98330 | 1 | 1 | 14.408097 | 7.2647 |
| **(c)** | 6 |  | 0.7781 | 2.55 | 0.425 | 2.731017 | 3.5099 |

**Table 2.1.2:** Measurements and metrics for sub-questions in Question 2 at **ϵ = 10**. All response times have been averaged over 3 time trials, each using separate executions of the program. The column “No Opt.” refers to executing the program without compiler optimization

**Question-2(e):**

Yes, as seen from the above data tables, the optimized algorithm is observably faster than the brute-force algorithm - both in sequential and parallel versions (with and without the -O3 optimization). As the total number of distance calculations is significantly reduced for each data point (as explained in answering Question-2(a) above), we observe the demonstrated performance gains. I tried increasing the total number of threads beyond the total number of physical cores available for the parallel version of the optimized algorithm to double-check any performance gains but did not observe any. However, in parallelizing the algorithm, using the **reduction** clause on the variable storing the total number of points within epsilon distance slightly improved the response time of the algorithm.

**Question-2(f):**

As detailed in Question-2(a) section above, the optimized algorithm significantly reduces the number of distance calculations by using a modified binary search algorithm to determine the total number of neighboring points within epsilon distance for any data point on the 2D cartesian plane.

**For the brute-force algorithm**, each data point has to perform **N** number of distance calculations and floating-point calculations (for the sake of analytical simplicity, we consider the number of floating-point calculations in calculating the distance between two 2D cartesian points to be one). As a result, for **N** number of data points, the total number of distance calculations (and floating-point calculations) will be **N2**.

**For the optimized algorithm,** let us make the following assumptions:

**N** => The total number of Point2D objects

**Nx\_unique** => The total number of unique x-values in the given data-set

**Ny (==** N) **=>** The total number of y-values in the data points

Let, the average number of neighboring Unique\_X objects within epsilon radius for each Unique\_X object = **neighbor\_x\_unique**

So, the total number of distance calculations per data point on the 2D grid on average is **(neighbor\_x\_unique)** [*finding out where the circle intersects the neighboring vertical lines as shown in Illustration-1*]

and, the total number of floating-point operations associated per data point on the 2D grid on average is 2\* **(neighbor\_x\_unique)**

[*calculating lower and upper limits of y as given by the intersecting points as shown in Illustration-1*]

The total number of distance calculations for all the data points will be **(Ny) \***  **(neighbor\_x\_unique)** instead of **N2** - a significant reduction on average. The total number of floating-point calculations also decreases similarly (**(Ny) \* 2** **(neighbor\_x\_unique)** instead of **N2**).

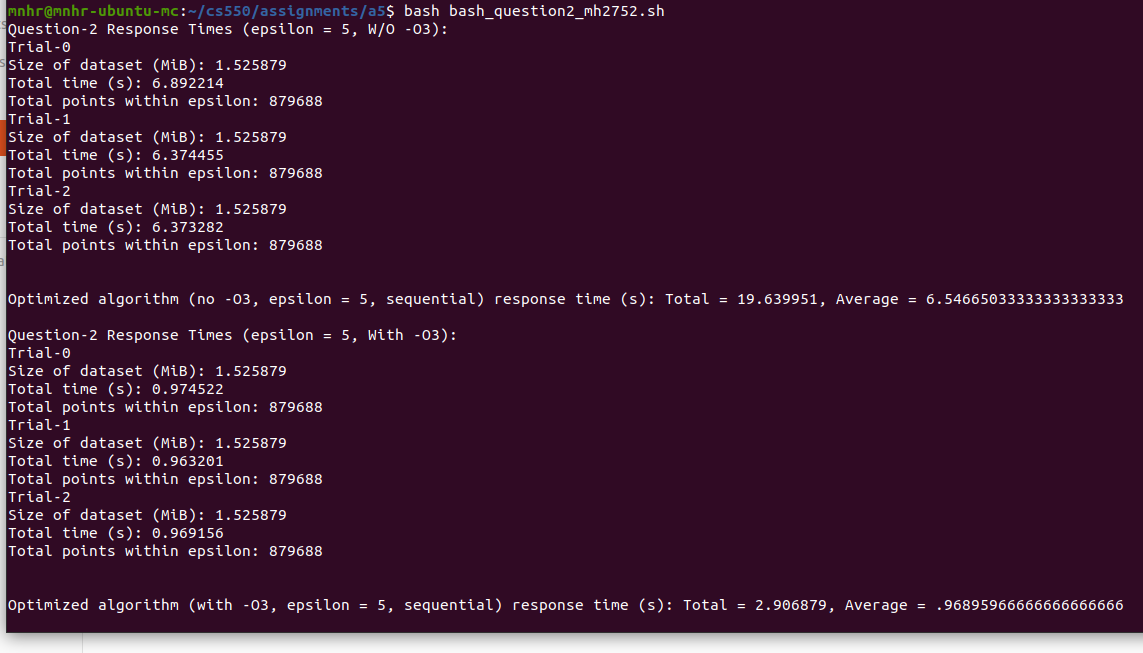
On average, when the unique x-values in the given data set are spread across a wider range than the epsilon value, **neighbor\_x\_unique** will be much lower than **Ny** as well as **N** (**neighbor\_x\_unique << Ny** and **neighbor\_x\_unique << N**). This further decreases the total number of distance calculations and floating-point operations as given by the equations **(Ny) \***  **(neighbor\_x\_unique)** and **(Ny) \* 2** **(neighbor\_x\_unique)** respectively.

**Moreover,** when **Nx\_unique ≅ N**, the average number of y-values associated with each Unique\_X object will be approximately 1 - thus the modified binary search algorithm (mentioned in Question-2(a) section above) will always execute in near-constant complexity, further increasing the performance of the optimized algorithm.

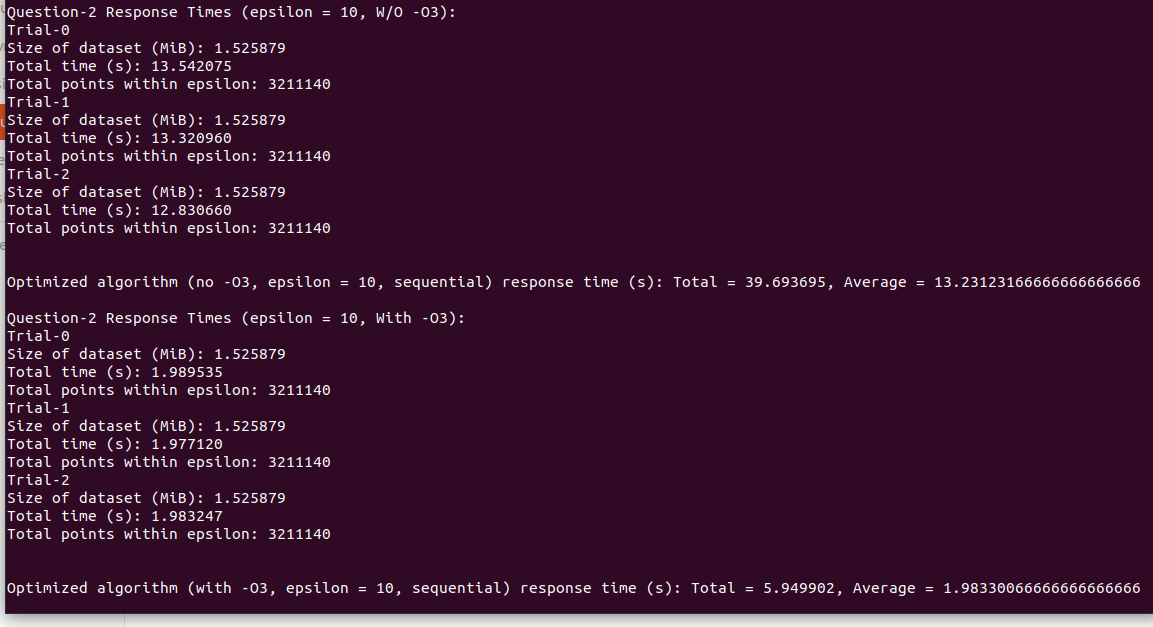
**Question-2(g):**

Yes, compared to the brute-force algorithm, the ratios of the response times of the corresponding sequential and parallel versions translate almost linearly in terms of epsilon values (halving the value of epsilon almost halves the brute-force vs. optimized algorithm response time ratios). As stated earlier, the total number of distance calculations and floating-point operations for each data point are governed by the value of epsilon and the total number of neighboring unique x-values within the epsilon radius. As the data points were the same with only varying values of epsilon in the program runs, halving the epsilon value halves the radius of the circle which essentially means the total number of neighboring Uniqu\_X objects are approximately halved - hence half the number of total distance calculations and floating-point operations on average.

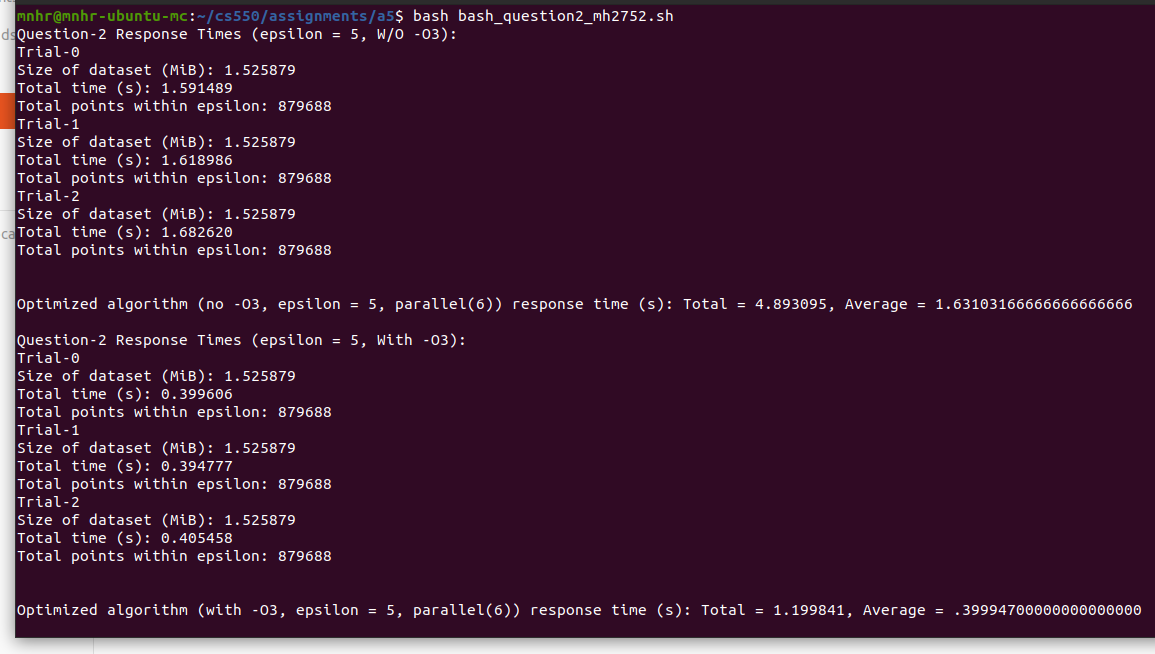
**Screenshots for Question-2 (Version: 1-Nazmul(mh2752))**



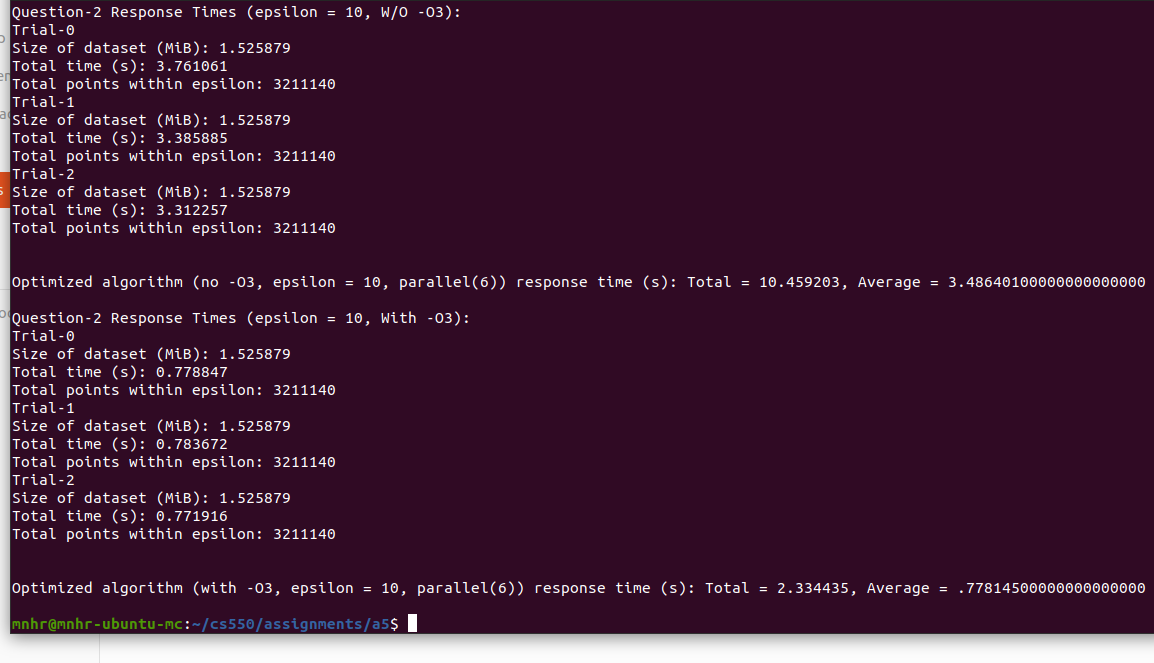
**Fig 5:** Optimized solution (***sequential***) response times **(top)** without using -O3 and **(bottom)** using -O3 for epsilon = **5**, N = **100,000** (Average of 3 independent runs)



**Fig 6:** Optimized solution (***sequential***) response times **(top)** without using -O3 and **(bottom)** using -O3 for epsilon = **10**, N = **100,000** (Average of 3 independent runs)



**Fig 7:** Optimized solution (***parallel***) response times **(top)** without using -O3 and **(bottom)** using -O3 for epsilon = **5**, N = **100,000** (Average of 3 independent runs)

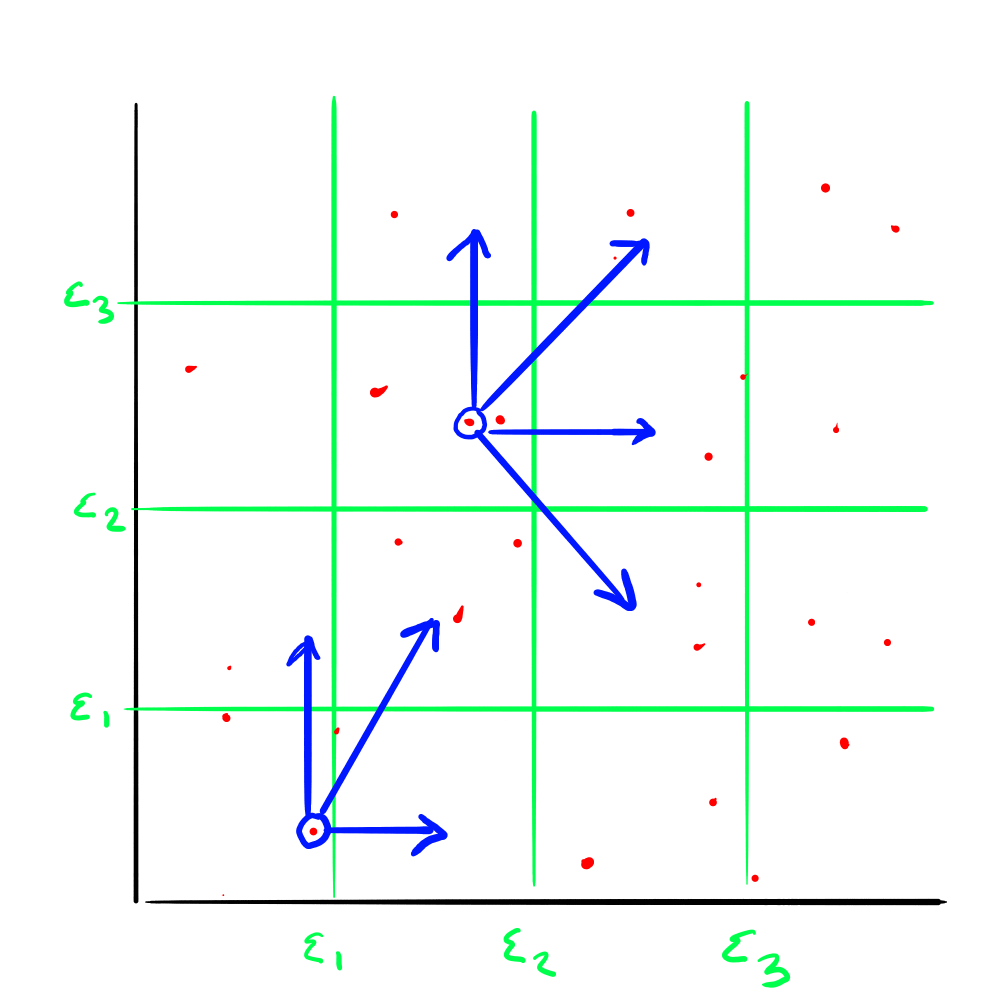


**Fig 8:** Optimized solution (***parallel***) response times **(top)** without using -O3 and **(bottom)** using -O3 for epsilon = **10**, N = **100,000** (Average of 3 independent runs)

**Ans. to Question-2(Version: 2-Felicity [fhe2])**

**Question-2(a):**

This approach uses an ε distance grid to order each of the data points into its respective x and y bin. This solution utilizes these bins to limit the number of distance calculations of each point to only points within it’s bin or an adjacent bin. Furthermore, points will update their neighboring points’ count so that each point may only need to look forward. Visualization of this solution is shown in **Fig. 9**.



**Fig. 9**: ε distance grid. Two points shown looking forward.

*Data Overhead*

In this solution, the struct for the data points was updated to include ints for binx, biny, and count, where binx and biny refer to their respective bin placement and count refers to the number of points within epsilon of the point.

The overhead for creating the x and y bins requires two scans of N (the number of points in the dataset) and one scan of nbins where:

In the first scan of N, the data points are assigned their bins and each x and y bin is sized to the number of data points in that bin. That number is recorded in the BinsSize array for its respective axis. Then the program scans the BinsSize array (length = nbins) to create a positions array (xBinsPos and yBinsPos) to determine the start of each bin in the respective x and y bin array. Finally the point ID, or the ordered number of the data point within the dataset is added to the x and y bins inside its assigned bin. An example of the array structure for the x bins is shown in **Fig. 10**.



**Fig. 10:** Example of x bins structure and traversal.

*Calculations and Parallelization*

By using the array structures described above, each point can directly access it’s neighboring points using their respective position in the xBins and yBins arrays. The program runs as follows:

#Parallelizer

For each point (N):

Check my x bin

For each point in my x bin or my x bin+1

If current point’s y bin is equal to my y bin or my y bin±1

Calculate the distance

If the distance is less than or equal to epsilon

#Atomic operation

Update my counter and the current points counter

The code is parallelized using dynamic scheduling around the outermost for loop so that each point can be checked in parallel. An atomic flag is added around the counter to avoid a race condition if two points are updating the same neighbor point’s counter at the same time.

**Question-2 (b), (c), (d):**

| **Subquestion (ϵ = 5 )** | **#Cores** | **Time (s) No Opt.** | **Time (s) w/ -O3** | **Speedup** | **Parallel Efficiency** | **Corresponding Brute-force\* Time (s)** | **Brute-force\* Vs. Optimized Algo. Ratio T1/T2**  **(d)** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Sequential | 1 | 0.798 |  | 1 | 1 | 474.425 | 594.518 |
| **(b)** | 6 | 0.204 |  | 3.912 | 0.652 | 84.546 | 414.441 |
| Sequential | 1 |  | 0.245 | 1 | 1 | 15.527 | 63.376 |
| **(c)** | 6 |  | 0.095 | 2.579 | 0.430 | 2.729 | 28.726 |

**Table 2.2.1:** Measurements and metrics for sub-questions in Question 2 at **ϵ = 5**. All response times have been averaged over 3 time trials, each using separate executions of the program. The column “No Opt.” refers to executing the program without compiler optimization

| **Subquestion (ϵ = 10 )** | **#Cores** | **Time (s) No Opt.** | **Time (s) w/ -O3** | **Speedup** | **Parallel Efficiency** | **Corresponding Brute-force\* Time (s)** | **Brute-force\* Vs. Optimized Algo. Ratio T1/T2**  **(d)** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Sequential | 1 | 1.64 |  | 1 | 1 | 474.794 | 289.509 |
| **(b)** | 6 | 0.373 |  | 4.397 | 0.733 | 82.292 | 220.622 |
| Sequential | 1 |  | 0.475 | 1 | 1 | 15.561 | 32.76 |
| **(c)** | 6 |  | 0.169 | 2.811 | 0.469 | 2.709 | 16.03 |

**Table 2.2.2:** Measurements and metrics for sub-questions in Question 2 at **ϵ = 10**. All response times have been averaged over 3 time trials, each using separate executions of the program. The column “No Opt.” refers to executing the program without compiler optimization

\*Brute force times here are different from the first question since these times were run on the same machine as this solution and the previous question 1 times were not. This is done for a more accurate comparison of the solution speedup.

Without optimization at ε = 5, this solution can run with over a 500x speedup compared to the brute force solution. However, with optimizations and as ε increases the speedup greatly decreases to about a 32x speedup. When both the brute force and this solution are parallelized the resulting brute force time is 16x slower than this solution.

**Question-2(f):**

The solution greatly decreases the number of distance calculations performed by only calculating points in the current or neighboring bins. The brute force solution calculates 100K2 distances which equals 10 billion calculations. This solution only performs about 1.2 million distance calculations for ε = 5 and 4.5 million calculations for ε = 10.

The overhead is negligible since creating the bin structures only required two scans of N. This means however, that this solution will not have a very high parallel efficiency since only the calculation section is parallelized and the work done by the calculation section is greatly reduced. It was not worth additional parallel overhead to parallelize the creation of the bin structures, again because the creation only requires two scans of N.

**Question-2(g):**

This program does about 8,333x (10B/1.2M) and 2,222x (10B/4.5M) less distance calculations for ε = 5 and ε = 10 respectively, and is 32 and 16 times faster than the brute force solution. As ε increases the speedup decreases since the number of calculations increases. When ε = 10 the program does a little under 4x as many calculations as when ε = 5. Since ε is a radius, the area of ε (Aε) is such that Aε = πε2. Meaning that if ε is multiplied by 2 then the resulting area will be 4x as large. Therefore, the number of distance calculations is likely to be 4x as many. With this in mind we can say that the number of calculations will increase and the speedup will decrease linearly with the size of ε. However, there is overhead in both the bin structuring, traversing the bins, and the parallelization, which is why the speedup is not directly reflective of the reduction in the number of distance calculations; i.e. the speedup will not be 2,222x faster for ε = 10 because of overhead.

**Screenshots for Question-2 (Version: 2-Felicity (fhe2))**

|  |  |
| --- | --- |
| a (Q2-E5-T1) | b (Q2-E5-T6) |
|  |  |
| c (Q2-E5-T1-O3) | d (Q2-E5-T6-O3) |
| **Fig. 11:** Question 2 - a) Fhe2 Solution, Epsilon = 5, No Optimization, Sequential, b) Fhe2 Solution, Epsilon = 5, No Optimization, 6 Threads. c) Fhe2 Solution, Epsilon = 5, Optimized with -O3, Sequential. d) Fhe2 Solution, Epsilon = 5, Optimized with -O3, 6 Threads | |

|  |  |
| --- | --- |
| a (Q2-E10-T1) | b (Q2-E10-T6) |
|  |  |
| c (Q2-E10-T1-O3) | d (Q2-E10-T6-O3) |
| **Fig. 12:** Question 2 - a) Fhe2 Solution, Epsilon = 10, No Optimization, Sequential, b) Fhe2 Solution, Epsilon = 10, No Optimization, 6 Threads. c) Fhe2 Solution, Epsilon = 10, Optimized with -O3, Sequential. d) Fhe2 Solution, Epsilon = 10, Optimized with -O3, 6 Threads | |

**Ans. to Question-2: Comparison of Versions**

Since this assignment was done in a group two solutions were created independently for question 2.

The first solution (version 1) used a repeated intersection check to compare each point to all points within ε on the x-axis, in doing so the search space was reduced. The second solution (version 2) also reduces the number of distance calculations but uses an ε grid to bin each of the points so that they only check their own and adjacent bins.

The first solution speedup over the brute-force was a high of 80x for ε = 5, non-optimized, sequential and low of 3.5x for ε = 10, optimized, on 6 threads. The highest run time for version 1 was 13.2 seconds on ε = 10, non-optimized, sequential. The lowest runtime was 0.399 seconds on ε = 5, optimized, 6 threads. The average parallel efficiency for this solution was 53.2%. This information is shown in Table 2.1.1 and Table 2.1.2.

The second solution had a high speedup over the brute-force of over 500x on ε = 5, non-optimized, sequential, and a low of 16x on ε = 10, optimized, on 6 threads. The highest run time was on ε = 10, non-optimized, sequential at 1.64 seconds, and the lowest was on ε = 5, optimized, 6 threads at 0.095 seconds. The average parallel efficiency is about 57.1%. This is shown in Tables 2.2.1 and 2.2.2.

From these results, we can say that version 2 outperformed version 1 in run time, speedup, and parallel efficiency overall. We have concluded that this is due to overhead since the number of distance calculations in total is similar between the two programs. Both solutions have overhead but the repeated calculations performed for version 1 significantly increased the runtime of the program compared to the traversal overhead of version 2.

**Bonus 1: Beat the Professor**

Please consider **Version 2** (solution 2 of question 2) for the beat the professor bonus points.

|  | T1/T2 (-O3) ε = 5 | T1/T2 (-O3) ε = 10 |
| --- | --- | --- |
| Sequential | 63.376 | 32.76 |

**Bonus 2: Competition**

Please consider **Version 2** (solution 2 of question 2) for the competition bonus points.