CS550 – Assignment 5

Question 1

The brute force algorithm is fairly straight forward. It uses two for-loops to go through every possible pair of coordinates and calculates the distance between those two coordinates. If that distance is less than ε, the "distances_within_epsilon" is increased by one.

The parallel version simply splits up the first for-loop between the threads with an "#pragma omp parallel for" command. Also, whenever "distances_within_epsilon" is incremented, a "#pragma omp atomic" command is used, to prevent multiple threads from incrementing at the same time.

a) & b) Below is a table of different tests that I have run on the algorithm:

ε	#Threads	Time, No Opt. (s)	Time, O3 (s)	Speedup (No Opt.)	Parallel Efficiency (No Opt.)	Speedup (O3)	Parallel Efficiency (O3)
5	1	560.047	20.286	1	1	1	1
5	2	323.155	12.147	1.733	0.8665	1.67	0.835
5	4	242.689	10.17	2.308	0.577	1.995	0.4987
10	1	561.681	20.295	1	1	1	1
10	2	322.386	12.718	1.742	0.871	1.596	0.798
10	4	244.26	10.213	2.299	0.5747	1.987	0.4967

c) As can be seen, the algorithm without the O3 compiler optimization takes quite a while to run, even when using multiple cores. The algorithm without O3 also does not have very good speedup or parallel efficiency, being barely twice as fast when using four threads, when we would expect it to run close to four times as fast.

With O3, the running time is much lower than without, being almost 30 times faster on a single thread. The speedup and parallel efficiency when using O3 however is worse than without it. With a parallel efficiency below 0.5 for four threads, the algorithm definitely does not scale well with multiple threads.

Question 2

a) In the optimized version of the algorithm, I take advantage of the fact that the distance between two points p1 and p2 is guaranteed to be more than ε when $|p1.x - p2.x|$ is more than ε.

I start by sorting my dataset according to the x coordinate, so coordinate with the smallest x is first, and the coordinate with the largest x is last. The sorting of the dataset contributes some overhead to the algorithm, and depending on the sorting algorithm used, this overhead could cause problems. I used the qsort() function from the standard C library, which is quite fast, and thus does not usually have a big impact on the running times of the program. Once the sorting is done, we go through the possible pairs of coordinates with two for-loops, as shown in the code below:

```
#pragma omp parallel for private(i, j, p1, p2) shared(data, epsilon, epsilon_square, distances_within_epsilon) schedule(dynamic)
for (i=0; i<N; i++) {
  p1 = data[i];

  for (j=i+1; j<N; j++) {
    p2 = data[j];

    if (p2.x - p1.x > epsilon) {
      break;
    } else if (euclid_dist(p1, p2) < epsilon_square) {
      #pragma omp atomic
      distances_within_epsilon += 2;
    }
  }
}
```

As can be seen, the code is not too different from the brute force algorithm. There are some key differences however:

Since the dataset is sorted according to the x coordinate, once we reach a value for p2.x in the second loop, such that $p2.x - p1.x > \varepsilon$, we can safely break out of this second for loop, since for every subsequent value of p2 in the loop, it will also be true that $p2.x - p1.x > \varepsilon$. Furthermore, we know that p2.x will always be larger or equal to p1.x, since the dataset is sorted, which is why we subtract p1.x from p2.x, and not the other way around. This saves us a lot of computation, especially when ε is small compared to the possible distances.

Additionally, we do not calculate every distance between the points, rather, we only calculate half of them. This is seen in the second for-loop, where j starts at i+1, rather than 0. This prevents us from "double counting", that is, it prevents us from calculating $d(p2, p1)$, when we have already calculated $d(p1, p2)$. We still want to count these "doubles" as part of the total distances however, and so we simply increment the "distances_within_epsilon" variable by two instead of one, every time a distance is found to be lower than ε. We also want to count the distances between a point and itself, but since we know that there will always be N of these distances, and since we know that these will always be less than ε (because the distance between a point and itself is 0), we can just add N to the total count after the algorithm is done.

b) & c) Below is a table of different tests that I have run on the algorithm:

ε	#Threads	Time, No Opt. (s)	Time, O3 (s)	Speedup (No Opt.)	Parallel Efficiency (No Opt.)	Speedup (O3)	Parallel Efficiency (O3)
5	1	2.643	0.112	1	1	1	1
5	2	1.524	0.079	1.734	0.867	1.412	0.706
5	4	1.166	0.066	2.266	0.566	1.697	0.424
10	1	5.271	0.195	1	1	1	1
10	2	2.985	0.14	1.766	0.883	1.39	0.695
10	4	2.295	0.109	2.297	0.574	1.789	0.447

On the table we see roughly the same trends as we saw for the Brute Force algorithm. The O3 version runs a lot faster than the non-O3 version, but the O3 version has worse scalability than the non-O3 version.

d) Below is a table comparing the Brute Force (BF) algorithm run times to the Optimized (Op) algorithm run times:

ε	#Threads	BF time, No Opt. (s)	BF time, O3 (s)	Op time, No Opt. (s)	Op time, O3 (s)	BF time / Op time (No Opt.)	BF time / Op time (O3)
5	1	560.047	20.286	2.643	0.112	211.9	166.28
5	2	323.155	12.147	1.524	0.079	212.043	153.76
5	4	242.689	10.17	1.166	0.066	208.138	154.09
10	1	561.681	20.295	5.271	0.195	106.56	104.076
10	2	322.386	12.718	2.985	0.14	108.002	90.84
10	4	244.26	10.213	2.295	0.109	97.717	93.697

e) As can be seen, my new optimized algorithm is much faster than the brute force algorithm, more than 200 times faster in some cases. We also see however that BF time / Op time becomes less and less the lower ε is. This is due to the fact that a smaller ε will lead to the statement $p2.x - p1.x > \varepsilon$ being true for a larger part of the coordinate combinations, allowing us to avoid a larger number of computations. Also, of note, the more computations we avoid, the more of an effect the sorting process will have on the total computation time. The sorting process takes ~0.025 seconds for $N = 100{,}000$, meaning that the sorting process can account for a significant amount of the total computation time, as is the case where we use 4 cores (as the sorting is not done in parallel), O3, and $\varepsilon = 5$, in which case the sorting accounts for around 30% of the total computation time.

I initially tried just adding a simple conditional variable, shown in the code below:

```
if (fabs(p1.x - p2.x) < epsilon && fabs(p1.y - p2.y) < epsilon && euclid_dist(p1, p2) < epsilon) {
  #pragma omp atomic
  distances_within_epsilon++;
}
```

The idea was that we could avoid the full distance calculation if we could observe beforehand that the distances in the x or y dimensions were already more than ε. This approach only caused a small decrease in computation time however (about 30%), probably because checking if distances in the x or y dimensions were larger than ε added a bunch of extra calculations that largely cancelled out the time saved not having to do the full distance calculation. In the final version, the only extra calculation is the p2.x – p1.x part, and it saves us a lot more time than it consumes.

f) The large increase in computation speed for my optimized version is due to the algorithm avoiding a large number of computations with the $p2.x - p1.x > \varepsilon$ condition. The smaller the ε, the larger the number of avoided computations will be. Thus, if we want to use a large ε, the computation time might be much larger, since we avoid a lot fewer computations. Thus, the largest factor to the increase in computation speed is that we avoid having to calculate the distance between a large number of coordinate pairs.

Another thing that saves time in my program is the fact that I got rid of the square root in the Euclidean distance function, and just compare the Euclidean distance to a pre-defined ε^2 instead of ε. This way, we avoid doing some floating point operations, which saves some computation time, although not a lot, as the program runs only about 10% faster with this addition.

The algorithm could conceivably be made to run even faster, if we were to take cache usage into account too, although I have not done so this time around.

g) I would say that these metrics translate roughly linearly to the observed reduction in response time. The two algorithms have roughly the same speedup and parallel efficiency, meaning that the increase in speed is not due to the optimized version being more efficient, but rather because the optimized version saves a lot of time avoiding calculations.

Bonus 1 & Bonus 2

I would like to try for the Bonus 1 and Bonus 2 questions. In the table below are the results of comparing my optimized algorithm to the Brute Force algorithm, both run with a single thread and with O3 compilation:

Threads	BF time / Op time (O3) $\varepsilon = 5$	BF time / Op time (O3) $\varepsilon = 10$
1	166.28	104.076

Also, shown below are example runs of both algorithms, along with their output:

```
frederik@DESKTOP-DLL5K0V:/mnt/c/Users/Gantzel/Desktop/Parallel_Computing/Assignment_5$ ./question1_fvg6 10

Size of dataset (MiB): 1.525879

Total time (s): 20.100697
total number of distances within ε: 3211140

frederik@DESKTOP-DLL5K0V:/mnt/c/Users/Gantzel/Desktop/Parallel_Computing/Assignment_5$ ./question2_fvg6 10

Size of dataset (MiB): 1.525879

Time to sort (s): 0.023365
Total time (s): 0.193895
total number of distances within ε: 3211140
```