Nearest and Furthest

Algorithm Design - Naive

In Serial

The standard naive algorithm in serial takes points in the form of two lists of x and y coordinates. Keeping track of the running best for the nearest and furthest point, for each point it iterates through the other points - calculating the distance to every other point and updating the running best when necessary. Note that we just keep track of the squared distance. Since the last step of the Pythagorean theorem is to take the root of the sum of the two squared distances (in x and y), we can be more efficient by taking the root once at the end, and use the squared distance for comparison. Furthermore, rather than using the pow() function for squaring, we use e.g., dist = x * x. Replacing all uses of pow took serial execution of 100k points using standard geometry from 319.3 seconds to 103.8 (on average). This is because the pow() function involves lots of unnecessary work - under the hood it uses the natural log and exponent so it's capable of handling fractional and negative powers, which is unnecessary here for squaring. Overall, this implementation has a time complexity of O(n^2).

By using a distance calculation function (`calcFunc`) as a parameter, we can calculate distances using standard Euclidean distance or by using the wraparound geometry, depending on the passed function. This approach requires no overhead, compared to if we'd used a flag of e.g., `useWraparoundGeometry`, then checking that flag at each distance calculation.

The distance calculation function takes 3 parameters - the first two are the difference between the two points in the x and then the y dimension. The third parameter is a pointer to an array - rather than returning the result of the distance, it places the distance in the given array. Since wraparound geometry can choose when to wrap around depending on if the aim is to maximise or minimise the distance, the function must return a closest possible and furthest possible distance for any given pair of x and y differences. In order to use the dependency injection, the calculation function for standard Euclidean geometry must do the same - except it just places the same distance in each of the points.

Once it's checked every other point, it adds the final best for the nearest and furthest distance to a vector, one for nearests and one for furthests.

Finally, once the nearest and furthest distances have been calculated for each point, we iterate through the list of nearest and furthest distances to calculate the mean for each. I experimented with calculating a running mean, using the following update (and likewise for furthest, not shown here);

```
avgNearest = (avgNearest * i + runningNearest) / (double) (i + 1);
```

However I found that running this update for each point was intensive, and by looping through them all at the end we see slight speed ups.

In Parallel

The parallel design is almost identical. However, we use a parallel for loop to split the work between as many threads as we have on the machine we're running on. This is achieved with an OpenMP parallel for loop. We set the result array `res` to be private to each thread - i.e., each thread has its own copy of `res`. This prevents race conditions occurring where multiple threads are writing to the same variable at the same time, then later reading from it. This would cause some data loss where a thread is accessing the wrong numbers. We could use a critical region for this instead, but we'd see massive slow down as threads would have to wait for a long time to enter the region.

Finally, to calculate the mean we again use a parallel for loop, with a sum reduction directive so that each thread calculates its own sum, before adding them all together once they've finished their work. Then we divide the sums by the number of points to calculate the mean.

Algorithm Design - Faster

In the previous design, we check each distance twice. For a point i, as it iterates through it will check every other point, including j. Then when we're finding the nearest and furthest distance for point j, it will again calculate the distance to i. One way of avoiding this would be to store the calculated distances in a hashmap, but we quickly find this to be foolish by realising that for 100,000 points, this would involve a 85GB hashmap in memory! Obviously this is unacceptable.

Instead, we realise that we can avoid this double calculation by keeping track of the running best nearest & furthest distances for all points at the same time. Rather than iterating through every point for each point, as we look at point i we only check point j such that j > i (j and i being the index of the point). Then, since we're storing the running best nearest and furthest for every point, once we've calculated the distance between a pair of points we can check whether the distance is an improvement on the running best for either point. This still has a time complexity of O(n^2), but with a smaller constant coefficient.

The rest of the design is the same, including the use of the distance calculation function and the calculation of the mean at the end.

Modifications in Parallel

Firstly, we use a dynamic schedule on the parallel for loop. Each iteration over i will have different amounts of work to do, since it only iterates over j for j > i. At first glance, I thought a guided schedule would be correct here, because the workload is triangular decreasing - i.e., for i=0 we process n-1 points, for i=1 we process n-2 points, … , for i=n-2 we process 1 point, and for i=n-1 we process 0 points. Guided scheduling gradually decreases chunk size, whereas

dynamic scheduling uses fixed chunk sizes. However, after investigating, I found that a dynamic schedule with a small parameter (~32) saw about a 5% speedup. I suspect this is because the irregular critical region access between iterations affects the time taken per chunk more than what the increased overhead from using a dynamic schedule does.

Once we've calculated the distance, we then compare the distance to the running best nearest and furthest distance. This involves checking several conditionals. At first glance, we can just put a critical region around the whole conditional checking part - however this again introduces a lot of waiting around for each thread. So we use a better pattern.

```
if (res[0] < nearests[i]) {
    #pragma omp critical
    {
        if (res[0] < nearests[i]) {
            nearests[i] = res[0];
        }
    }
}
```

This is an example of checking if the shorter distance (res[0]) between point i and j is an improvement on the running best for point i (at nearests[i]). We do a first check on it, called a stale read - most iterations will not involve updating the running best, so it will rarely enter the critical region. Then, once inside the critical region, it checks again - this prevents a situation occurring where two threads pass the conditional at once, as they've both found an improvement - however they write to the nearests vector in the order such that the worse improvement writes over the better improvement. For example, if the current closest distance is 0.5, and two threads separately find a distance of 0.3 and 0.4, but the thread with 0.3 writes first, followed by the thread with 0.4 - so after the whole sequence, it stores 0.4 as the closest, rather than 0.3. By checking again once inside the critical region, we get a read that's guaranteed to be accurate, and in the example above would correctly result in 0.3 being stored as the closest.

Speed Analysis

Each of these numbers is the mean time taken (in seconds) across 10 simulations of 100k randomly initialised points. For fair comparison, each of the 10 simulations had a unique set of randomly initialised points, that each implementation of the function worked off of. These simulations were run on a MacBook Pro, M1 Max chip, with 10 native threads.

Geometry	*Algorithm*		
		Serial	**Parallel**
Standard	**Naive**	103.8	12.91
Wraparound	**Naive**	315.0	38.73
Standard	**Fast**	76.70	10.98
Wraparound	**Fast**	174.8	23.48

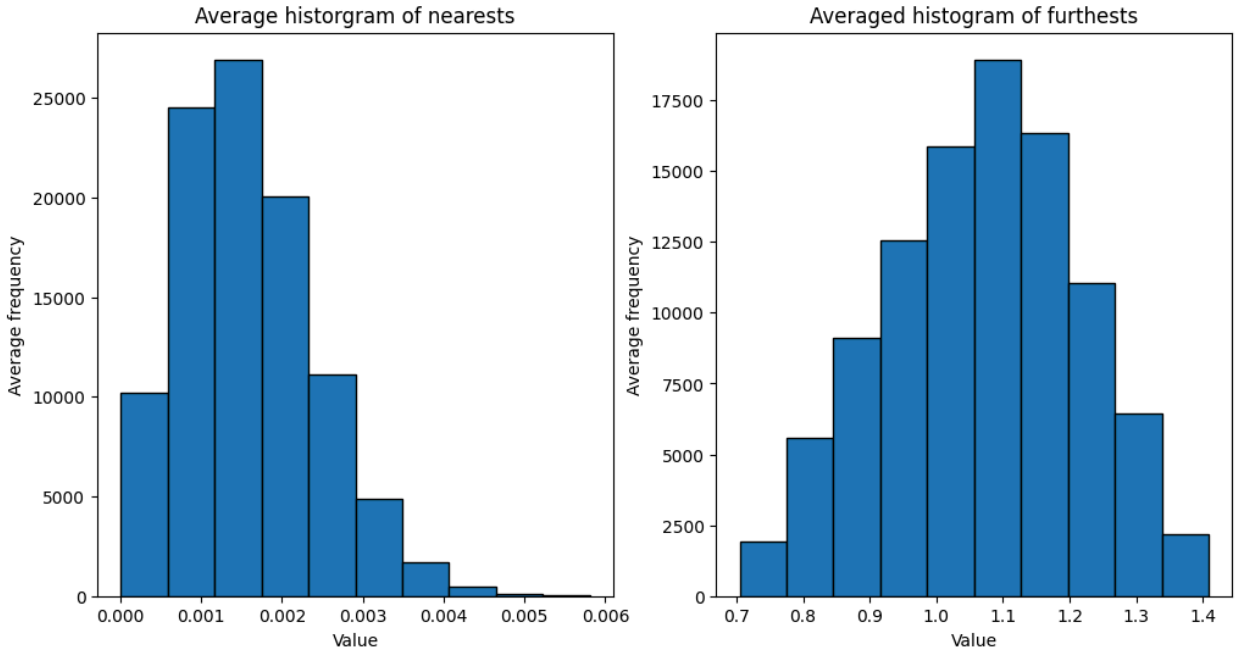
A quick takeaway is to note that the parallel times are all just under 10x faster, for the most part. This reflects the number of threads used - 8 of the threads are designed for Performance, and 2 for efficiency. Combined with overhead introduced by parallelisation, this explains why the parallel times aren't exactly 10 times faster.

Distance Distributions

Here, the fast parallel versions of the function have been used for all calculations. Each graph is for 100k points. Again 10 simulations were run, and averages taken.

Using Standard Geometry

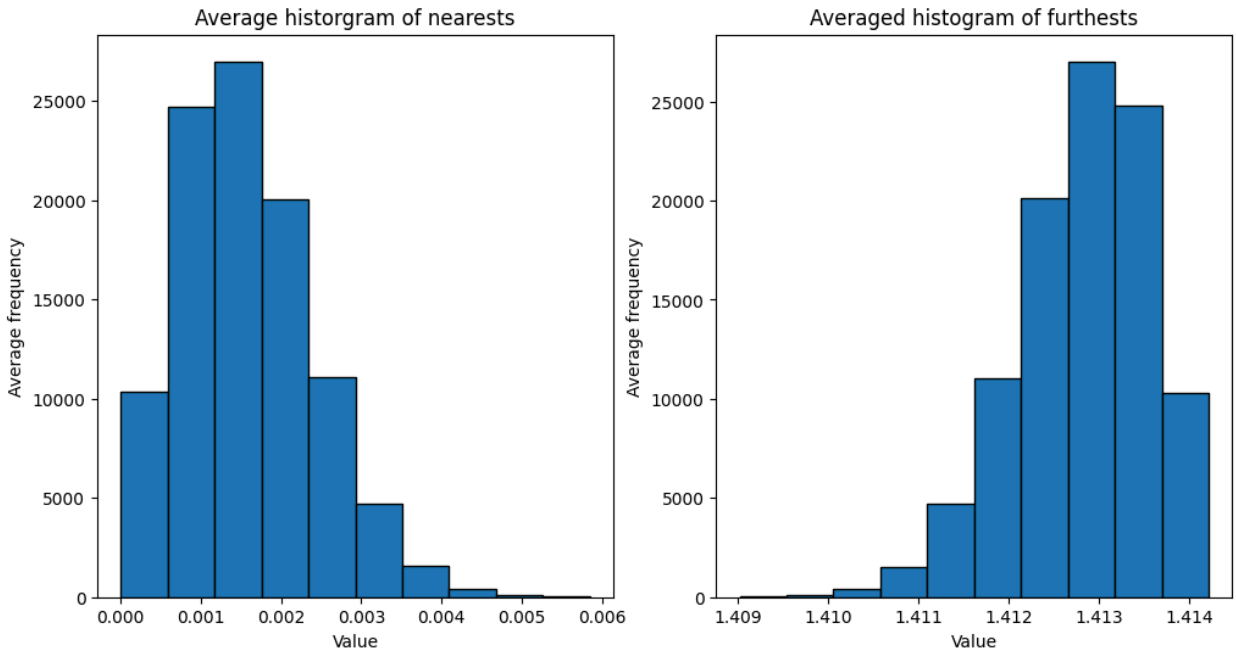
I found a mean nearest distance of 0.00158207, and a mean furthest distance of 1.067968.



The nearest distances are very right skewed, all being very close to 0 - this makes sense considering there are a lot of points in a very small area. The furthest distances are more evenly shaped, around 1.1. This is because the maximum furthest distance is different for each point - a point right in the middle could not have a point more than $1/\sqrt{2}$, whereas a point in the corner could have a point in the opposite corner as the furthest away point - giving it a distance of $\sqrt{2}$.

Using Wraparound Geometry

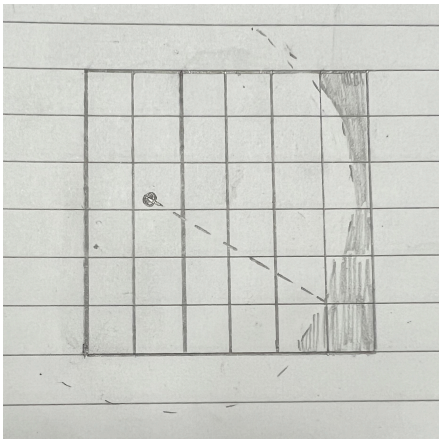
I found a mean nearest distance of 0.00158203, and a mean furthest distance of 1.412812.



The distributions here are due to the wraparound geometry giving the effect of all points being mirrored on each side of the 1x1 box they are initialised in. The nearest distances are affected very little by wraparound geometry, since it'll only give a different nearest point (compared to standard geometry) for points that are extremely close to the boundaries. Also, we notice that the distributions are almost symmetrical - this is because the nearest point in standard geometry is also often the furthest point in wraparound geometry.

A different algorithm - the grid method

If we were solving this by hand, we'd just look by eye at the closest ones, and then measure the distances, and likewise for the furthest points. We can achieve something similar by preprocessing the points into a grid. To find the nearest point to any point, we only look in a point's own cell and its neighbouring cells. To find the furthest point, I've implemented it here such that we check all cells around the edges. This could be improved though.



The furthest possible a point can be away is in the furthest away corner. The algorithm already assumes that each cell has at least 1 point in, so the following steps would result in a faster method.

1. Find the distance to the nearest corner of the furthest away corner.
2. Creating a circle with that radius, find the cells that the edge of the circle directly passes through (i.e., cells that have any amount of them outside the circle).
3. Only search through those cells for the furthest away point.

Here, that would be in each of the cells that have any shading in. This works because we know that each cell has at least one point inside, and the theoretical furthest away point would be in the furthest away corner. However, we don't know where inside that corner cell the point will be in, so we assume the worst that it is right on the corner. So there could be a further away point, in a different cell - that area of possible point locations is represented by the area outside the circle.

This grid method results in massive speedups, taking on average 29.30 seconds for 100k points. Furthermore, if we take out the code for calculating the furthest distances, it takes only 1.51 seconds, compared to 103.8 seconds for the naive serial (this calculates the furthest distance as well, but taking that out would hardly affect the runtime as it only involves a conditional, there's no extra computation). This huge speedup occurs because the grid method speeds up finding the nearest neighbour a lot more than finding the furthest neighbour. In terms of Big O notation it has O(n) pre-processing time, then O(n sqrt(n)) search time.

Conclusion

There are many ways of calculating the nearest and furthest neighbour. The most interesting method I found that could be used is a k-d tree, which has significant pre-processing work, but then for the actual search of a nearest neighbour is really quick - much faster than any of the algorithms used here.

There's also almost certainly more ways we can speed up the implementations here, whether that's using clever compiler flags, or neat tricks in the mathematical parts.