Exploring Parallel Processing for Nearest and Furthest Point Calculations in Datasets

Nathan Truong

November 27, 2024

1 Introduction

1.1 Outline of Problem

On a 1.0 x 1.0 square, at least 100,000 points were generated in random positions inside the square. From the system, the main objective the program had to fulfill was to calculate and store a point's nearest and furthest distance to another point in two geometries. The distribution of distances to then be plotted and compared. The first geometry was the standard geometry. Where the distance between points were calculated using Pythagoras' theorem as usual. The second configuration used wraparound geometry, which allows for shorter distances when points are placed at opposite ends of the square. In this geometry, if two points are far apart, the distance is calculated by "wrapping" around the edges of the square, effectively bringing the points closer together by traveling across the boundary. This results in a shorter path compared to standard geometry. This is demonstrated in figure 1 These distance calculations was performed for all the points generated in the square with both geometries.

1.2 Parallelisation - Thread Allocation

With an objective of generating at least 100,000 points, this would mean at minimum, a total of 9,999,900,000 distance calculations would be performed. An additional investigation was to determine the optimal configuration for thread allocation. Two approaches were considered: the first involved using all available threads to calculate all distances in standard geometry first, followed by calculating all distances in wraparound geometry. This sequential method ensured that all resources were dedicated to one geometry at a time. The second approach split the available threads, allowing the distances for standard and wraparound geometry to be calculated concurrently. This concurrent method aimed to balance the workload across

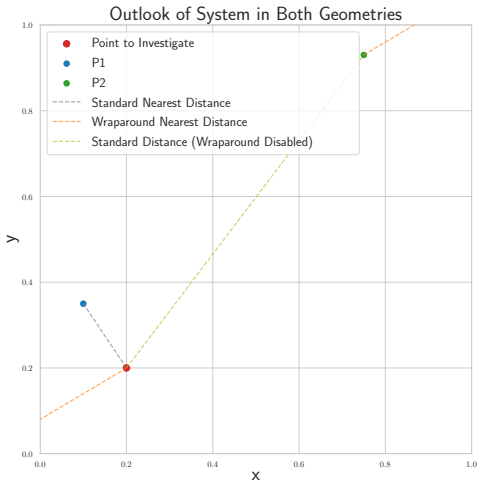


Figure 1: Illustration of geometries and system of 3 points. Notice how the closest point from red to blue is within the standard geometry. However the distances between red to green is varied between geometries. The length of the orange line in the wraparound geometry is smaller than the olive line.

threads and potentially improve efficiency by performing both calculations simultaneously. The names allocated to both configurations was "all" and "partitioned" respectively during development.

2 Methodology

The investigation was performed with a main C++17 program with all visualisations created with python modules, Seaborn and Matplotlib.

2.1 Distance Calculations

2.1.1 Standard Geometry

A class that represents a point was created. From this, a vector that stores all 100,000 randomly generated points was created. Once the vector gets filled with point objects, the distance between a point and all other adjacent points is calculated, with the smallest and largest distances being stored in separate vectors. This process is then repeated for each subsequent point. Afterward, the smallest and largest distances were saved to separate files. Using the vectors that stored these distances, the corresponding averages were calculated and then output.

2.1.2 Wraparound Geometry

The method for calculating the distances in the wraparound geometry was nearly identical to the standard geometry. However before the distance was calculated, the difference in x and y between two points was checked to have exceeded half the length of the square (0.5). If this was the case, the distance was adjusted by adding 1 to wraparound. The distance calculations then proceeded as normal. The files were labeled with wraparound geometry in mind to prevent confusion.

2.2 Parallelisation - OpenMP

To perform the thread allocation investigation, the Open Multi-Processing application programming interface (OpenMP) was used. This came with several advantages apart opening the opportunity to use more than one logical processor (thread). One of the key advantages of using OpenMP is its design for portability [1], which enables code sections to be scheduled for parallel execution in a specialised manner. This feature allowed the parallelisation of distance calculations in both geometries, while ensuring that race conditions were avoided thanks to the reduction and critical clauses. For the purpose of the investigation, the main scheduler used to calculate the distances in both geometries and calculate the respective averages was static. This approach is the easiest to understand, as it allocates iterations to each thread in a straightforward manner. For a given iterator, each thread is assigned a specific subset of equal iterations to execute [2]. As a result, the static scheduler has minimal overhead. This approach also works well with when each thread has the same workload since each thread is performing the almost the same algorithm with minimal wait times.

To allocate threads to work on calculating distances in both geometries sequentially, the functions that performed the calculations were simply called in sequence. To allocate threads concurrently, the *sections* construct was used [3]. The functions that calculate

distances in the standard geometry and wraparound geometry were placed in their own OpenMP *section* (under the sections construct).

To investigate the effects of different thread allocation configurations, the total time taken to complete all four calculations, the nearest and furthest distances in both geometries, was recorded. For both thread allocations, this was varied with a quantity of threads specified. Which can be changed with a command line input and automated further with a shell script.

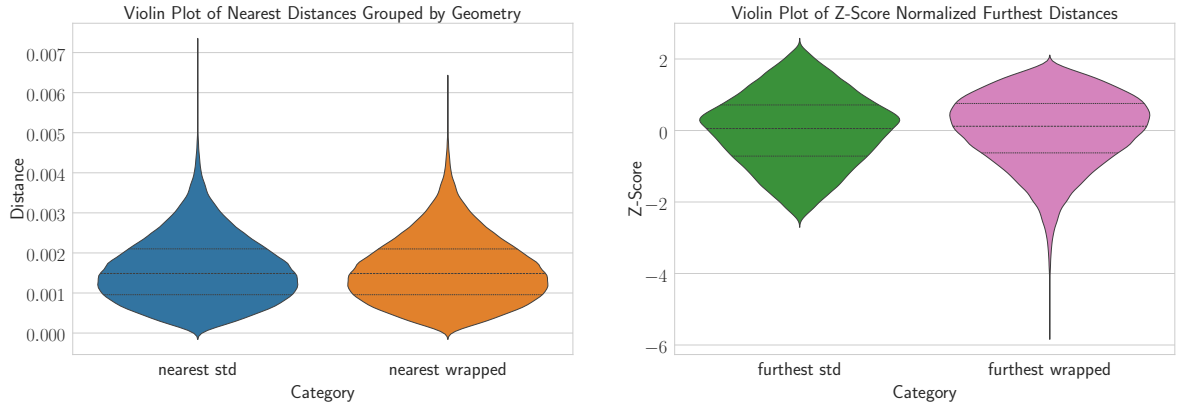
3 Results and Discussion

3.1 Distribution of Distances

Violin plots were plotted to investigate the distribution of distances. For the nearest distances, the distribution of distances in both geometries had approximately the same shape. The maximum nearest distance measurement under the wrapped geometry is smaller compared to standard. This is likely because of the wrapped geometry enabling an extra "movement option" for calculating distance. Nevertheless, the average nearest distance measured in both geometries does not vary much away from 0.00158. More precisely, the percentage difference was found to be $> 0.5\%$. For the furthest distances, it can be seen that the violin around the median for the wrapped geometry is rounder than the plot for the standard geometry. This indicates that it is more probable to obtain a furthest distance close to the median distance under the wrapped geometry. Whereas, in the standard geometry, the probability drops more linearly around the median. It is however worth noting that figure 2b uses standardised measurements between geometries calculated using z-scores. The scale of the furthest distances differs significantly between the two geometries, showing a percentage difference of $\approx 34\%$.

Geometry	**Avrg. Nearest Distance**	**Avrg. Furthest Distance**
Standard	0.00158451	1.06909
Wraparound	0.00158238	0.705709
Percentage Difference	**0.13443%**	**-34.02161%**

Table 1: Average Nearest and Furthest Distances for Different Geometries, with Percentage Differences



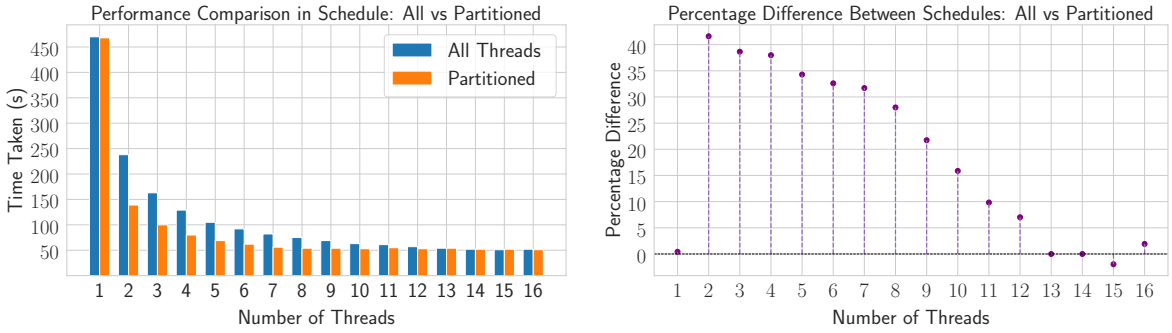
(a) Nearest distances.

(b) Furthest(z scored) distances.

Figure 2: Violin plots of the nearest and furthest distances in both geometries.

3.2 Calculation Run Time and Parallelisation

The time taken to measure the nearest and furthest distances and calculate the average in both standard and wrapped geometries decreased as the amount of threads used increases in both thread allocation options.



(a) Time comparison. (b) Percentage difference between thread alloc'.

Figure 3: Comparison plots of run time with number of threads used. The processor used was an AMD Ryzen 7 5700X.

From figure 3 the gains in partitioning the threads between calculating distances in both geometries extremely evident between using 2 to 10 threads. The percentage difference in run times compared to dedicating all threads to one geometry is $\approx 15\%$ from using 2 to 10 threads. However for both thread allocation options, upon using more than 12 threads, it is visibly clear that there is a point of diminishing returns. This is likely due to increased overhead in allocating threads tasks from using more threads [4]. Additionally, using more threads can increase synchronization overhead for critical operations, such as file writing, as the likelihood of longer wait times for thread synchronization grows.

4 Conclusion

The distribution of measured nearest distances in a standard geometry versus a wrapped geometry is almost identical. The wrapped geometry having a lower maximum due to points being more likely to take a wraparound approach. This in turn reduces the maximum distance between points within the 1.0 x 1.0 square. This conclusion could also illustrate that a big distribution of distances captured for the wraparound geometry indicate distances almost equivalent to the standard geometry. The distribution of measured furthest distances in both geometries is roughly the same as well. However, from the median, there is a steeper drop off in the standard geometry than the wrapped geometry which indicates higher variability in furthest distances in the standard geometry. Additionally, the scale between both geometries furthest distances is different by a much larger margin compared to the nearest distance in both geometries. This is also due to the wraparound geometry allowing the distance calculation to effectively use another method of movement. Once again in turn, decreasing the maximum distance between points.

The effect of using all threads to sequentially calculate distances in both geometries against calculating distances concurrently through partitioning the threads both observe reductions in run time. However calculating the distances concurrently has been proved

to generally reduce run times to a greater effect with less threads. There is also a point where using a certain amount of threads does not gain any more reductions in run times. The reason for this is increased overhead from managing more threads.

References

[1] OpenMP Architecture Review Board, *Overview of the OpenMP API*, *OpenMP Application Programming Interface Version 6.0*, Nov 2024, p. 2. Available online: `http://www.openmp.org/mp-documents/spec30.pdf`.

[2] OpenMP Architecture Review Board, *schedule Clause*, *OpenMP Application Programming Interface Version 6.0*, Nov 2024, p. 418. Available online: `http://www.openmp.org/mp-documents/spec30.pdf`.

[3] OpenMP Architecture Review Board, *sections Construct*, *OpenMP Application Programming Interface Version 6.0*, Nov 2024, p. 407 - 409. Available online: `http://www.openmp.org/mp-documents/spec30.pdf`.

[4] Mark Roth, Micah J Best, Craig Mustard, and Alexandra Fedorova, *Deconstructing the overhead in parallel applications*, **2012 IEEE International Symposium on Workload Characterization (IISWC)**, pp. 59–68, 2012, IEEE.