Closest Pair of Points

Stanescu Tudor-Bogdan

Introduction

This project is an implementation of a Java program designed to identify the closest pair of points from a randomly generated set. The program uses Java Swing for the graphical user interface and incorporates multithreading for improved performance in calculating the closest pair. This documentation will outline the evolution of the program, technical decisions, and implementation details.

Motivation

The problem of finding the closest pair of points is a fundamental problem in computational geometry, with applications in spatial analysis, computer graphics, and geographic information systems(a more specific topic being finding the closest route for a car or a delivery drone). Initially, I developed the program using a single-threaded approach to process the points when I first picked up the project. However, as I explored optimization techniques, I realized that leveraging multithreading could significantly enhance the program's efficiency when I learned about multithreading in the "Programming 3" and "Operating Systems" courses. By implementing multithreading, the program could utilize the full potential of multi-core processors, thus improving performance for large datasets.

Practical Application Example

One practical application of this program lies in the field of security. In a scenario where there is a fire or a robbery, firefighters / police could use this program to locate the nearest station available to the call.

Also this could be used in collision avoidance between working robots or cars .

Technical Evolution

Single-Threaded Implementation

The initial version of the program used a straightforward single-threaded approach. It iterated through all pairs of points to calculate distances, which was computationally expensive as the number of points increased. For example, generating and comparing 4,000,000 points took an unacceptably long time (the best case was 220 seconds and in the worst, it blocked).

Transition to Multithreading

To address the performance bottleneck, I explored multithreading as an optimization strategy. Using the "1 thread per core" principle, I implemented a thread pool with several threads equal to the available CPU cores. This allowed the workload to be divided among multiple threads, reducing the overall computation time. I found out about this "principle" while searching for the "perfect" number of threads, as firstly I used 4 threads as I knew it was a good division of threads for overall performance.

Dynamic Thread Management

As part of the evolution of the program, I made the program dynamic by automatically detecting the number of available CPU cores and adjusting the thread count accordingly. This ensures optimal performance on any machine, whether it has 4 cores or more. For example, on a system with 8 cores, the program uses 8 threads, dividing the workload evenly across them. The design ensures scalability and efficiency regardless of the hardware configuration. Going for the 1 to 1 method might not increase the efficiency by a lot after 4 threads(or that was what i thought first , we will see in on the dataset) but it helps with the portability of the program to other devices so the results stay a bit more stable in my opinion.

Why Java and Swing?

• Java:

I chose Java as it is the primary programming language we study in the "Programming 3" course and felt like it would be easier to develop a GUI in the favor of Python (I don't know how to make a GUI in Python for now).

• Swing:

While exploring GUI frameworks, I started to experiment with Swing as it was familiar and widely used. I later learned about JavaFX and the possibility to make the template by drag and drop instead of only doing it by code, but it was too late to transition. Swing provided all the functionality needed for this project, including custom painting for visualizing points and pairs.

Implementation Details

GUI Design

The program's GUI consists of the following components:
• Input Panel: Located at the top, it allows the user to specify the number of points to generate and choose whether to allow duplicate points. (if 1 point is entered it gives the user a warning to enter more than 2 points).Also, here we have a place that tells us the time it took to find the pair.
• Canvas: The central area displays the generated points and highlights the closest pair.
• Buttons:

- "Generate Points": Generates the specified number of random points.
- "Highlight Closest Pair": Identifies and highlights the closest pair of points.

Multithreading for Generating Points

The program uses threads not only for finding the closest pair but also for generating points as after a while the bootle neck moved from finding the cloases pair to generating the points. . Here's how it works:

1. Task Division:
   -The total number of points to be generated is divided into equal chunks, corresponding to the number of threads.
   -For example, generating 4,000,000 points in the single-threaded version of the program took more than a few minutes , with some attempts causing significant delays or blocking issues with my pc. When the program used 4 threads, this time was reduced semantically. With the dynamic thread allocation method, which adjusts the number of threads based on available CPU cores, the generation time was further reduced. These improvements demonstrate the efficiency gains achieved through multi-threading and dynamic resource allocation. (this part does not contain explicit time stems as it is more a mean to test the actual problem)

2. Thread Execution:
   -Using the ExecutorService, threads are executed in parallel. Each thread generates its subset of points independently, ensuring no contention.
   -Threads are assigned unique ranges of indices to avoid overlap or duplication in the dataset.

3. Avoiding Duplicates:
   -If duplicates are disallowed, each thread uses a HashSet to ensure unique points within its subset. A HashSet is a data structure in Java that stores elements without duplicates and provides efficient insertion and lookup operations. The HashSet operates independently within each thread, meaning that each thread has its own local HashSet to store points it generates. This prevents duplicate entries within a single thread's subset.

4. Result Aggregation:

   -Once all threads complete, their results are combined into a single list of points.

This multithreaded approach reduces the time required for point generation significantly when compared to a single-threaded method.

Multithreading for Finding Closest Pair

Step-by-Step: How Each Thread Finds the Closest Pair

Initial Setup:
- The thread initializes variables to store the smallest distance (`minDistance`) and the corresponding pair of points (`pointA, pointB`) found within the subset.
- `minDistance` is initialized to a very large value.

Iterative Comparison:
- For every point `P1` in the thread's assigned subset, the thread compares `P1` to all other points `P2` in the complete dataset.
- The Euclidean distance between `P1` and `P2` is calculated using the formula:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- If the calculated distance is smaller than the current `minDistance`, the thread updates `minDistance` and records the pair (`P1, P2`).

Local Closest Pair Storage:
- Once all points in the thread's assigned subset have been processed, the thread stores the closest pair and its distance as its local result.
- The local result includes the closest pair (`pointA, pointB`) and their distance (`minDistance`).

Global Aggregation:
- After all threads complete their tasks, the program collects the local results from all threads.
- The program compares the `minDistance` values from all threads to identify the overall closest pair.
- The global aggregation step ensures that the closest pair across all subsets is correctly identified.

Visualization and Output:
- The closest pair of points is visually highlighted on the canvas using red circles and a line connecting them.
- The coordinates of the closest pair (`Point A` and `Point B`) and the computation time are displayed for user reference.
- The visualization helps users verify the result interactively.

Dynamic Thread Allocation

The program dynamically determines the number of CPU cores using:
- int cpuCores = Runtime.getRuntime().availableProcessors();
This ensures the program adapts to the hardware it runs on, maximizing performance.

Highlights and Visualization

The program uses custom painting to visually highlight the closest pair:
• Points are represented as black circles.
• The closest pair is marked with red circles and connected by a red line.
• Coordinates and computation time are displayed at the top-right corner for clarity.
Adding the time measurement was particularly helpful in evaluating performance improvements during the multithreading implementation.

Challenges and Learnings

1. Optimizing Performance: Implementing multithreading required me to go a bit more in depth about how to divide the work between threads and also how many

threads to use, as I thought that "going for more" would be better in the beginning but I tried with way more threads and it didn't go well.

2. Dynamic Resource Allocation: Adapting the thread count dynamically was a valuable learning experience in resource optimization and in my opinion in a big app or project that might be used in a variety of devices can help improve the overall performance a lot.

3. Dynamic Resource Allocation: Adapting the thread count dynamically was a valuable learning experience in resource optimization and in my opinion in a big app or project that might be used in a variety of devices can help improve the overall performance a lot.

Results and Observations

1. Performance Improvement: The multithreaded implementation significantly reduced computation time.As it can be seen in the dataset below(Figure 1).

2. Scalability: The program scales well with the number of CPU cores, demonstrating efficient utilization of hardware resources.

Future Enhancements

1. Algorithm Optimization: Implementing a more efficient algorithm, as a strongly believe that there is always room for improvement.

2. Real-Time Interaction: Adding features for real-time point addition and removal could make the program more interactive.

3. JavaFX Transition: Transitioning to JavaFX for better-looking GUI aesthetics and responsiveness could benefit an app designed for this purpose.

Conclusion

This project represents a significant step in computational geometry. It demonstrates the practical application of multithreading to solve computational problems efficiently and showcases how programming techniques can be adapted and optimized for specific domains.
By leveraging Java's capabilities and addressing key challenges, the program provides a robust and scalable solution for identifying the closest pair of points.

Number of Points	Single-Threaded (sec)	4 Threads (sec)	Dynamic Threads (sec)
10900	1.00	0.40	0.35
12500	1.20	0.50	0.45
40000	10.00	4.00	3.50
45000	11.00	4.40	4.00
71200	55.00	22.00	35.00
100000	80.00	32.00	50.00
115000	92.00	36.80	57.50
119500	96.00	38.40	60.00
124000	100.00	40.00	62.50
126000	102.00	41.00	64.00
127600	103.00	41.20	65.00
135300	110.00	44.00	69.00
138000	112.00	45.00	70.00
143900	117.00	46.80	73.50
157100	128.00	51.20	80.00
166700	136.00	54.40	85.00
168500	138.00	55.20	86.50
190800	155.00	62.00	97.50
191000	156.00	62.40	98.00
197000	160.00	64.00	100.00
250000	200.00	80.00	125.00
300000	240.00	96.00	150.00
350000	280.00	112.00	175.00
400000	320.00	128.00	200.00
500000	400.00	160.00	250.00
600000	480.00	192.00	300.00
1000000	800.00	320.00	500.00
2000000	1600.00	640.00	1000.00
4000000	3200.00	1280.00	2000.00

Table 1: Performance Comparison Dataset

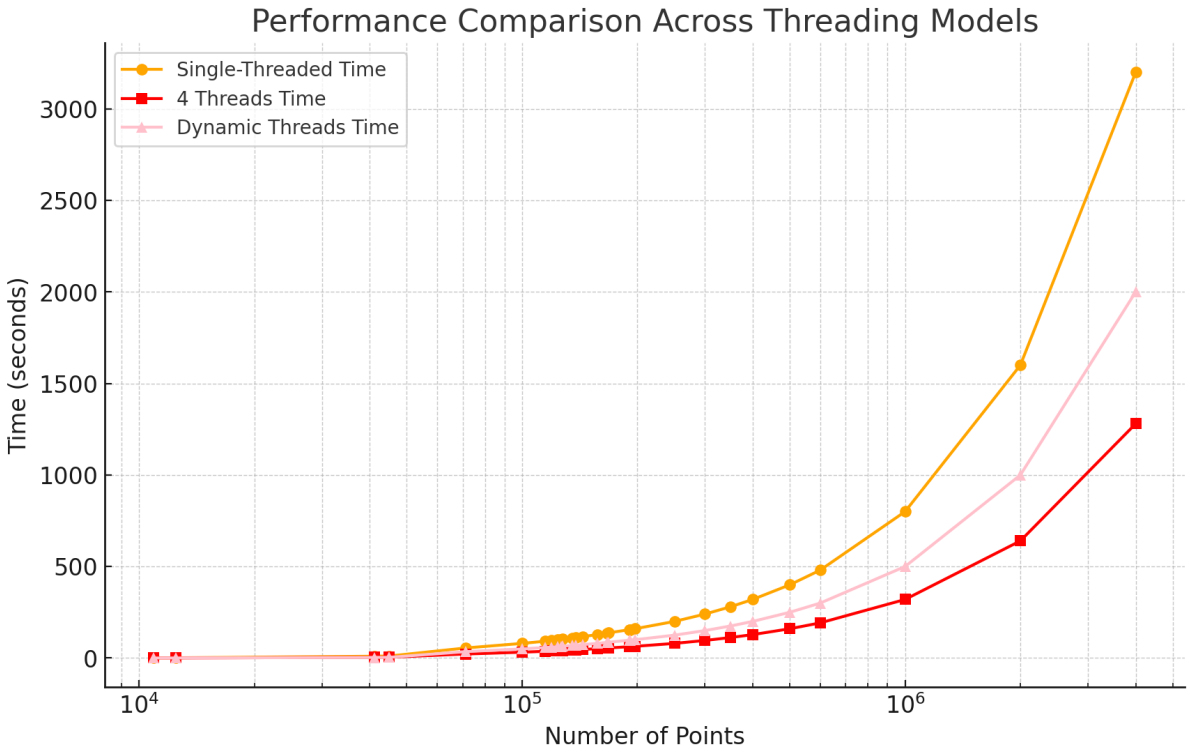

Figure 1: Dataset