**My computer:**

MacBook pro

13-inch, 2017

Processor: 2,3 GHz Dual-Core Intel Core i5

Memory: 8 GB

Cores: 4

**How to run program (in the oblig\_4 folder):**

Compile

* javac ConvexHull.java

Run

* java ConvexHull <n> <seed>

**Introduction**

This is a report about my sequential and parallel program for a convex hull and my findings about the timings of the program.

**Sequential program**

I used the algorithm described in the oblig paper.

**Parallel program**

My parallel implementation I believe is somewhat different than the algorithms that we have been shown. The parallelization is simply a split of workload to each thread where all threads work concurrently on their set of points, the program can either be set to have 2 or 4 threads where each thread works on finding the points on the convex hull for their size of the pointset. The best analogy is splitting the pointset to either 2 or 4 pieces of the same size depending on how many threads you want. Handling for correct order is done separately and sequentially after all threads are done running. The algorithm for finding the correct order in the parallel version is included in the timing as its running time is insignificant compared to finding all points on the convex hull.

I used semaphore as a synchronization method. Each thread waits for permits to add new points to the list of points on the convex hull whenever a new point is found. When a new point is added the thread releases its permit and lets a new thread access the same list. The list of points is located in the parent class ConvexHull.java.

**MEDIAN TIME (ms) & SPEEDUPS WITH 2 THREADS**

|  |  |  |
| --- | --- | --- |
| n | Seq (ms) | par (threads: 2) |
| 100 | 0,09 | 0,44 |
| 1000 | 1 | 2 |
| 10000 | 2,47 | 4,89 |
| 100000 | 17 | 11,8 |
| 1000000 | 171,9 | 85,62 |
| 10000000 | 1577 | 811,3 |

TIME

SPEEDUP

|  |  |
| --- | --- |
| n | speedup |
| 100 | 0,20 |
| 1000 | 0,50 |
| 10000 | 0,51 |
| 100000 | 1,44 |
| 1000000 | 2,01 |
| 10000000 | 1,94 |

**TIME (ms) & SPEEDUPS WITH 4 THREADS**

TIME

|  |  |  |
| --- | --- | --- |
| n | Seq (ms) | par (threads: 4) |
| 100 | 0,12 | 0,62 |
| 1000 | 0,99 | 2,79 |
| 10000 | 6,14 | 2,87 |
| 100000 | 16,12 | 10,50 |
| 1000000 | 175,00 | 75,50 |
| 10000000 | 1507,00 | 652,00 |

SPEEDUP

|  |  |
| --- | --- |
| n | speedup |
| 100 | 0,19 |
| 1000 | 0,35 |
| 10000 | 2,14 |
| 100000 | 1,54 |
| 1000000 | 2,32 |
| 10000000 | 2,31 |

As we can see there is speedup > 1 achieved for n > 1 000 000 in both versions. If we compare 2 and 4 threads, we can see that with 4 threads, we get a significantly larger speedup for n = 10 000. This is most likely a cause of the doubling of threads that leads to a bigger split in concurrent workload. Based on the graph I believe there is an appropriate extrapolation to say that the speedups become asymptotical at 2,5. This, and the fact that we observe a significant difference in speedup at medium sizes of n might be a symptom of an underlying weakness in my parallel implementation. The fact that I am separately finding the correct order in the convex hull sequentially might be the cause. When we get larger sets of points, the number of points on the convex hull increases and a separate algorithm that only handles the points on the convex hull might become significant and blunts further speedups. But anyway, I am satisfied with a speedup of 2 for n > 1 000 000.

**Discussion on Alternatives for Parallelizing the Convex Hull Problem**

In addressing the convex hull problem, our goal is to explore and assess alternative parallelization strategies that could potentially enhance performance over the current implementation which divides the workload among threads, processing subsets of points to identify hull candidates. This setup, while effective, encounters bottlenecks in merging results and ordering points correctly.

**Alternative Parallel Methods**

Multi-layered Thread Utilization:

Implementation: Initial threads manage data segments, potentially spawning sub-threads for complex or large segments.

Advantages: Dynamically adapts to workload variations, potentially improving resource utilization.

Challenges: Increased overhead from dynamic thread management.

Experimentation and Parameter Variation

To ascertain the effectiveness of these methods:

Experiment 1: Implement both strategies, varying thread counts (2, 4, 8, 16) across large datasets (up to 10 million points), and measure execution times.

Experiment 2: Record CPU and memory usage to gauge resource efficiency and scalability.

Performance Comparison

Metrics: Execution time, scalability, and resource efficiency will be benchmarked. The divide-and-conquer method is expected to excel with very large datasets due to effective task subdivision and merging. The multi-layered approach may benefit uneven data distributions.

Conclusion

This discussion has introduced robust parallelization techniques for the convex hull problem, offering potential performance improvements. By experimenting with these methods and adjusting thread parameters, we aim to identify optimal strategies that balance speed and resource management, crucial for processing extensive datasets efficiently. These findings will guide the development of algorithms that are not only theoretically sound but also practically superior.

N = 100

Seed = 42

A screenshot of a computer

Description automatically generated