Approximate k-Nearest Neighbors with Parallel Computing Techniques

Ioannis Fraidakis

November 11, 2024

Abstract

This research focuses on developing an approximate k-nearest neighbors (ANN) algorithm in C, with an emphasis on parallel computing techniques. The primary goal is to balance computational speed and accuracy when finding nearest neighbors in high-dimensional datasets. We implement and compare multiple parallel computing approaches, including OpenCilk, OpenMP, and Pthreads, to optimize the performance of the ANN algorithm.

1 Introduction

The goal of this work is to write C code for the approximate k-nearest neighbors (ANN) algorithm for a query set Q with respect to a corpus set C of dimension d, followed by optimization using parallel computing techniques. The approximate nearest neighbor search aims to balance speed and accuracy, especially when the exact k nearest neighbor (KNN) computation becomes computationally prohibitive due to large data volumes or high dimensionality.

We focus on the case where $C = Q$ (with the possibility of an easy extension to the general case). Ultimately, we aim to find the optimal trade-off between processing speed (queries per second) and accuracy (recall).

2 Proposed Algorithm

The algorithm recursively partitions Q into discrete subsets until each subset has a size of at most **MIN_SIZE**. For each such subset, we calculate the exact k-nearest neighbors and then "merge" the partial solutions.

Let $Q1$ and $Q2$ be two subsets to be merged (with $Q1 \cap Q2 = \emptyset$). The merging process is as follows:

- Select **sample_size** random points from $Q1$
- Find their exact k-nearest neighbors in $Q1 \cup Q2$ by calculating Euclidean distances with all points in $Q2$
- For the remaining non-sampled points in $Q1$, find the N closest sampled points
- Create a candidate set of points to examine (N*k in total)
- Examine each candidate point and update the k-nearest neighbors accordingly

3 Implementation

3.1 Sequential (Non-Approximate) Implementation

We begin by implementing the sequential, nonapproximate version (task V0) to establish a baseline for code and benchmarking. This allows for later comparison of exact results with approximate variants, both in terms of execution time and the number of correct neighbors found.

Note that distance calculations utilize OpenBLAS library functions, which run on multiple threads, while the remaining code remains sequential.

3.2 Parallel (Approximate) Implementation

We transform the sequential algorithm into an approximate version, starting with OpenCilk to easily identify parallelizable sections. This is due to automatic workload distribution and the guarantee of at most two times slower execution than the optimal parallel version, leveraging the work-stealing concept.

Using Linux's perf tool, we identify bottlenecks in the ANN implementation and stages that would benefit from multithreaded programming.

For each slow code point, we examine two parallelization techniques:

- Creating new threads with `cilk_spawn`
- Using `cilk_for` for parallel computations within loops

The results align with the following theory:

- Creating additional threads (`cilk_spawn`) is efficient only in recursive stages
- For other stages (exact calculations for sampled points and merging), using `cilk_for` for simultaneous point processing is more efficient due to lower overhead

3.3 Correctness Verification

We verified the correctness of the sequential implementation using precomputed data for the case where $C \neq Q$, confirming that the sequential algorithm produces expected results.

We then used this sequential implementation to compute the "correct" results for our studied case ($C = Q$), which formed the basis for evaluating parallel implementations.

To ensure the correctness of parallel versions:

- We started execution with a single thread and gradually increased thread count
- We observed that recall percentage remained stable, with only execution time variations
- This strongly indicates the absence of race conditions and unintended data overwrites

Each thread writes to separate memory locations, avoiding the need for synchronization with locks or mutexes and making the implementation simpler and more efficient. We confirmed these indications using ThreadSanitizer (TSan).

4 Optimization Techniques

4.1 Performance Parameters

We can control the accuracy-speed trade-off by modifying:

- **MIN_SIZE**: Determines the minimum subset size for processing
- **Sampling Reduction**: Controls the number of samples
- **Candidate Reduction**: Affects the number of sampled points to examine

We used the *perf* tool to identify cache misses and introduced prefetching instructions for data to be used in subsequent iterations.

For query set point storage, we used the `alignas` specifier to ensure data arrays are memory-aligned according to CPU SIMD (Single Instruction - Multiple Data) requirements.

5 Computational System

Operating System	Ubuntu 24.04
Processor	AMD Ryzen 5 5600H, 3301 MHz, 6 Cores, 12 Logical Processors
Cache Sizes	L1 = 384 kB $\|\|L2 = 3$ MB$\|\|L3 = 16$ MB
Total Physical Memory	13.9 GB

6 Benchmarks

6.1 Recall – Queries per Second Trade-off

Key observations:

- For high recall (>95%), all implementations show similar performance
- For lower recall (<50% for Fashion-MNIST, <80% for SIFT), OpenCilk excels due to lower thread creation overhead

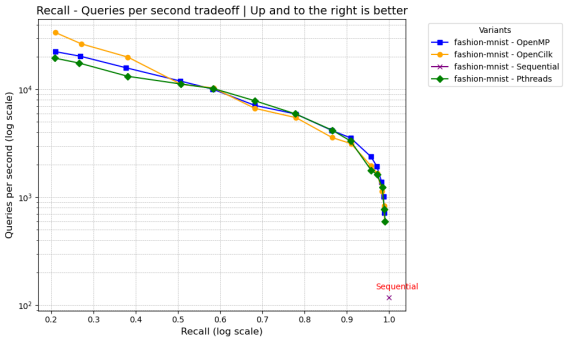

Figure 1: Performance for Fashion-MNIST dataset.

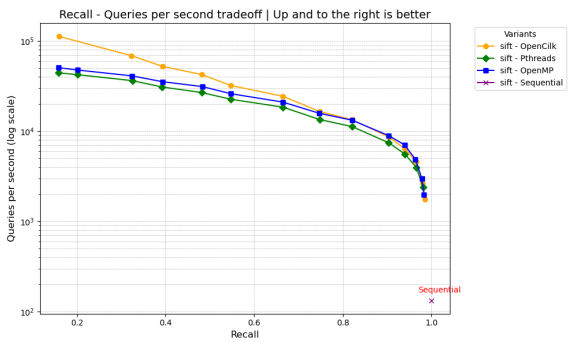

Figure 2: Performance for SIFT dataset.

6.2 Speedup Compared to Sequential Execution

Key characteristics:

- All implementations show similar speedup up to the number of logical processors (12 threads)
- OpenCilk outperforms in managing more than 12 threads, showing linear performance degradation instead of exponential
- No significant improvement observed at 14 threads

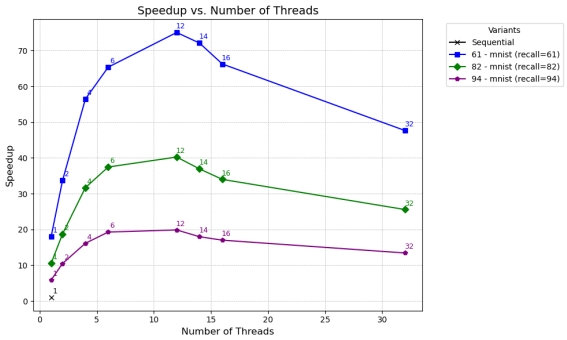

Figure 3: Speedup for all datasets.

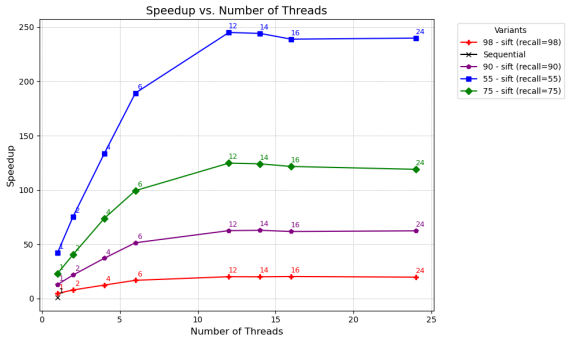

Figure 4: Speedup for the SIFT dataset.

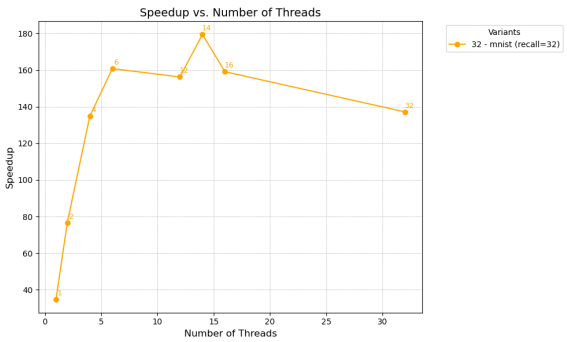

Figure 5: Speedup for all datasets.

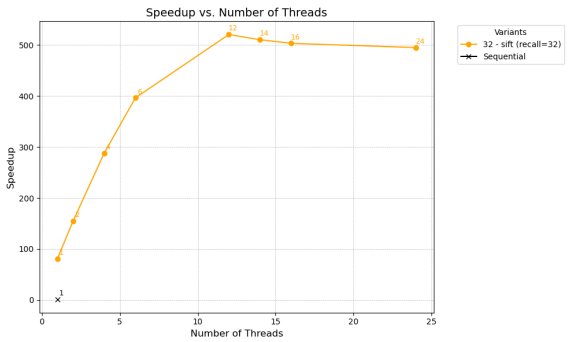

Figure 6: Speedup for the SIFT dataset.

For a satisfactory recall (>75%), we achieve up to 125x faster execution for the SIFT dataset, primarily due to drastically reduced computations for approximate neighbors.

7 Code Availability

The complete source code is available on GitHub at: `https://github.com/fraidakis/PDS_Ex1`