CPEN512 Project Report

Jiaqi Zhang (63174551)

1. Introduction

k-nearest neighbours (*k-NN*) is a classic algorithm first introduced in the field of statistics. In a typical classification scenario, we have a set of training samples with labels. When given an input sample and ask for its class membership, k-NN can be utilized to identify its nearest neighbours and thus infer its class accurately. The algorithm is widely used in many areas, such as text mining, stock price prediction, medicine and facial recognition.

2. Problem Setting

2.1 Description

In this project, we will be working on a slightly modified version of traditional k-nearest neighbours problem. We are given two sets of vectors of the same dimension, D. One set (ny) would be the base dataset (training samples), and the other (nx) is query set. Our goal is for each vector in nx find *topK* (K=5) nearest vectors in ny. The distance metric will be Euclidean distance. All vector has a label embedded to it, and we will perform classification tasks for query vectors. Labels are assigned by majority vote, i.e. the most frequently appearing label among topK results.

2.2 Dataset

We use two matrices A and B to represent nx and ny, respectively. Vector dimension will be set to 256, and the last element in each row is the label. We also set number of vectors in ny (row number of B) to be 128x that of nx, so if matrix A has a size of (N, 256), matrix B will have a size of (128*N, 256). Each element in the matrix is a randomly generated float between 0 and 1.

3. Implementation

Besides sequential version, we have implemented three different parallel versions for this project, including **pthread**, **Open MPI** and **OpenMP**. We will summarize the detailed implementation of each method (with possible optimizations) in this section, and share our experiment results in the next section.

3.1 Sequential

The sequential version is straightforward with three for-loops. The first loop is for matrix A (query) and second loop is for matrix B (base). For each vector pair, we calculate the Euclidean distance between them using a third loop on their dimension D. This is implemented in

getEuclideanDistance function shown in the code snippet. We store the query vector's distances with all base vectors, and then perform findTopK to obtain the indices of K smallest distances. Here we make an optimization - instead of sorting and getting the first K entries, we implement a quick select algorithm which runs in O(N) time. The quick select algorithm is to randomly choose pivots until pivot = K. For each pivot, partition the array into two parts. Unlike quicksort, which recurse in both sides, quick select only work on the side with the element we are searching for. This effectively reduces average time complexity from O(NlogN) to linear.

```
for(int i=0; i<M; i++) {
    for(int j=0; j<N; j++) {
        record[j] = getEuclideanDistance(A, i, B, j, D);
    }
    topK = findTopK(record, N, K);
    A[i*D+D-1] = labelVectorArray(topK, K, B, N, D);
}
```

Figure 1. Code snippet for sequential version

Another huge optimization we made in sequential implementation is to use an instruction level parallization technique - Streaming SIMD Extensions (SSE). Basically SSE is a CPU instruction set with 128-bit registers, each register capable of processing 4 single-precision floats simultaneously. This means that in Euclidean distance calculation $\text{dist} = \sum_{i=0}^{d}(x_i - y_i)^2$, using SSE could largely speed up computations. And indeed, as shown in the below graph, using SSE (on server, with -msse3 flag), we observe a 4x speed up when base vector count is more than 30,000. In fact, there might exist other faster instruction sets such as more recent AVX/AVX2/AVX512, but unfortunately they are not supported on the server, so we did not have a chance to experiment it.

Note that both quick select and SSE optimizations are applied to all below methods.

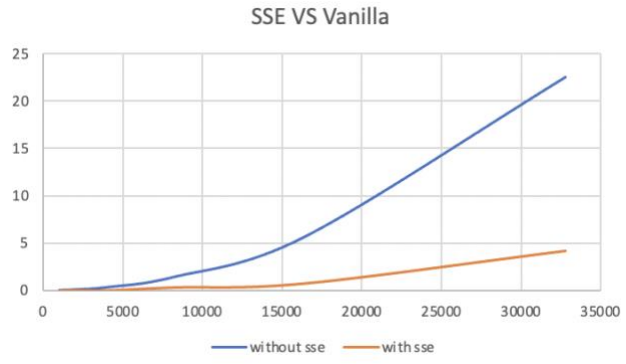

Figure 2. SSE Optimization on sequential version

3.2 pthread

The main idea of pthread implementation is to divide the small matrix into P (thread number) parts so that different rows of submatrices can be processed by the corresponding thread at the same time. The logics will be similar to that in Gaussian elimination. Barrier was applied to synchronize threads.

3.3 Open MPI

Similar to pthread, we also divide the matrix into P (thread number) parts so that submatrices can be processed simultaneously. Unlike pthread, we use message passing via MPI_Send and MPI_Receive to distribute rows, calculate vector distances and gather results.

3.4 OpenMP

In OpenMP, the code structure would be similar to that of sequential version k-NN. The main difference is we add pragmas to the for loop. Pragma is used to tell the compiler how to fork additional threads and process data enclosed in the loop in parallel. Firstly, we get all the distances between vectors from two sets of matrices by OpenMP. Secondly, topK nearest neighbours are selected and we label the vectors. The process to find topK and label query vector is also supported by OpenMP. With OpenMP, we can also generate matrices in parallel.

4. Results

In this section we will share our experiment results and compare performance among different implementations. (Matrix size refers to number of rows in matrix B)

4.1 Sequential

As we can see from the bar chart, runtime and matrix size follows a quadratic relationship, which is expected because if both matrix A and B has doubled their row count, total calculations would quadruple.

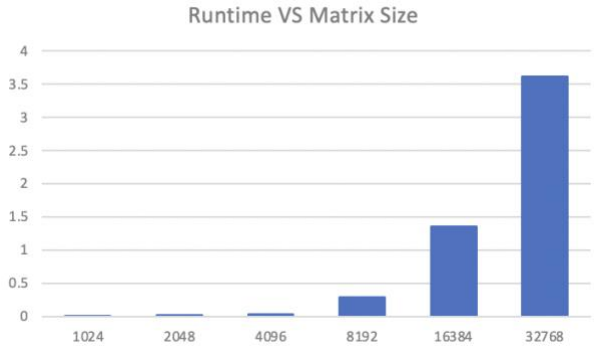

Figure 3. Runtime vs Matrix Size - Sequential

4.2 pthread

The performance of the pthread implementation has greatly improved as we increase the thread number. This is because in parallel programming, different processors can do different calculations at the same time, so when we use 32 threads (maximum in server), we observe the best speedup. Speedup is approximately 32x compared to sequential version when matrix size is 32768.

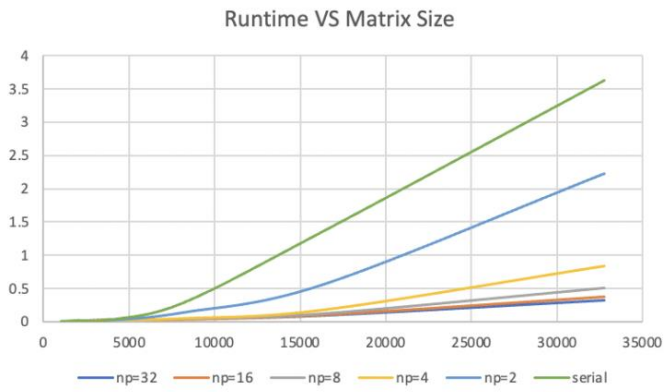

Figure 4. pthread VS sequential

4.3 MPI

The performance improves a lot by utilizing MPI. Resulting speedup is similar to that of pthread. The speedup is also proportional to number of threads used.

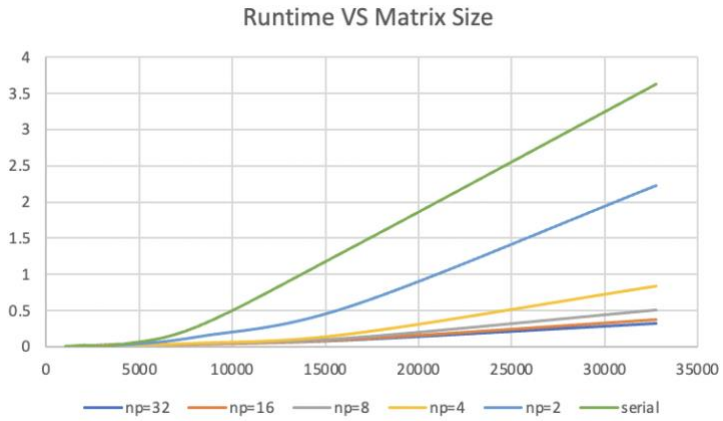

Figure 5. MPI VS sequential

4.4 OpenMP

The performance improves a lot by utilizing OpenMP. Speedup is similar compared to pthread and MPI. We will summarize their similarities below.

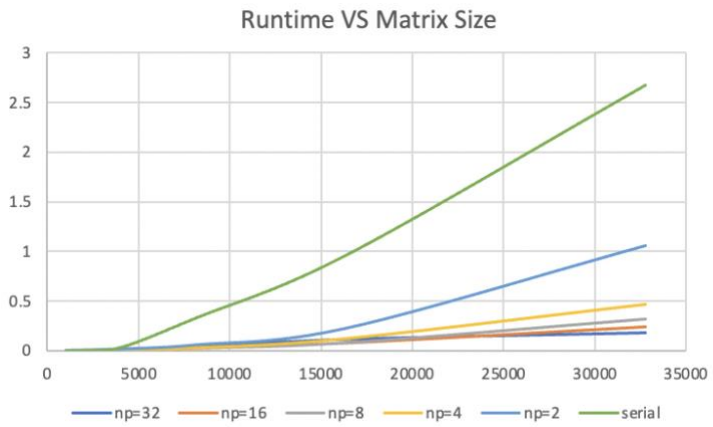

Figure 6. OpenMP VS sequential

4.5 Comparison between different methods

4.5.1 Similarities

1) OpenMP, MPI and pthread are all powerful tools for parallel programming. The basic intuition is similar by dividing large amount of work into smaller pieces and small parts can be processed at the same time.

2) The performances of three methods are quite similar based on test results, and the performances of three methods are highly related to the number of threads. Pick appropriate number of threads is crucial for the speedup.

3) Unlike CUDA and Vector Blox, Optimization Settings can greatly improve performances of these methods.

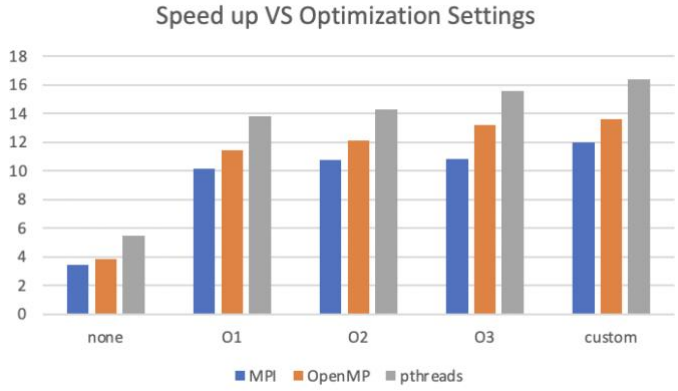

Figure 8. Speed up VS Optimization Settings

4.5.2 Differences

1) Pthreads and MPI are low-level APIs for working with threads. However, OpenMP is high level and a standard for application-level programming. OpenMP is a compiler based technique to create concurrent code from serial code.

2) The strategies to achieve their goals are different. In MPI, communications between groups are set up by sending and receiving messages. In OpenMP, parallelism is achieved by adding pragmas. In pthread, different flows of work are controlled by settings like locks and barriers. MPI is message based and OpenMP is derivative based.

3) Programming models are different. MPI is a programming model for distributed memory devices. This implies that parallelism occurs when each parallel process operates in its own memory region, apart from the others. However, OpenMP and Pthread are programming models for shared memory devices. This means each parallel thread has access to all data when parallelism occurs.

4) Drawbacks of pthreads:
    a) It is sometimes hard to find data racing bugs because they can be intermittent.
    b) Sometimes easy to encounter deadlocks. As such, OpenMP is often an alternative solution to pthreads in real world applications.

5. Conclusion

In this project we implemented k-NN with different parallelism techniques and compared their performance. Information on how to compile and run the program is listed below.

Validation: For validation purposes, we have compared the results of sequential k-NN with OpenMP, pthreads and MPI separately. The results turn out to be identical.

MPI Compile: mpicc -O3 -msse3 knnMPI.c -o mpi -lm
MPI Run: mpirun -np 32 ./mpi 32768 256 32
MPI Best Result: 0.323944 second

OpenMP Compile: gcc -O3 -fopenmp -msse3 knnOpenMP.c -o kmp -lm
OpenMP Run: ./kmp 32768 256 32
OpenMP Best Result: 0.264245 second

pthreads Compile: gcc -O3 -pthread -msse3 knnPthread.c -o pth -lm
pthreads Run: ./pth 32678 256 32
pthreads Best Result: 0.193495 second