Mina Naseh, Hesam Korki

Profiling Report for KNNClassifier

Overview

This report summarizes time spent in key functions of the `KNNClassifier` to identify performance bottlenecks.

Function Breakdown

`fit`

- **Purpose**: Stores training data (`X_train` and `y_train`).
- **Total Time**: 4 µs
  - `self.X_train = X`: 3 µs (75%)
  - `self.y_train = y`: 1 µs (25%)
- **Analysis**: Very fast, negligible impact on performance.

`euclidean_distance`

- **Purpose**: Calculates Euclidean distance between two points.
- **Total Time**: 1262.86 s
  - `diff = (x1 - x2)`: 163.98 s (13%)
  - `sqr_diff = diff ** 2`: 138.88 s (11%)
  - `sqr_diff_sum = np.sum(sqr_diff)`: 784.15 s (62.1%)
  - `return np.sqrt(sqr_diff_sum)`: 175.84 s (13.9%)
- **Analysis**: Time-consuming, especially `np.sum(sqr_diff)` (62.1%).

`predict`

- **Purpose**: Generates predictions by calling `_predict` on each point.
- **Total Time**: 1852.01 s
  - `y_pred = [self._predict(x) for x in X]`: 1852.0 s (100%)
  - `return np.array(y_pred)`: 96 µs (0%)
- **Analysis**: Most time is spent in `_predict`, indicating it as a performance bottleneck.

`_predict`

- **Purpose**: Computes distances to all training points and finds `k` nearest neighbors.
- **Total Time**: 1850.98 s
  - `distances = [self.euclidean_distance(x, x_train) for x_train in self.X_train]`: 1842.45 s (99.5%)
  - `k_indices = np.argsort(distances)[:self.k]`: 8.5 s (0.5%)
- **Analysis**: Distance calculations in Line 28 are the primary bottleneck.

Summary

- **Key Bottlenecks**: `euclidean_distance` and `_predict` dominate runtime, especially distance calculations.
- **Optimization Targets**: Lines 17 in `euclidean_distance` and 28 in `_predict` are prime candidates for parallelization.

Parallelization Strategy in `parallel_vs_sequential_knn.py`

This script implements a parallel version of the K-Nearest Neighbors (KNN) algorithm using a hybrid approach that combines **MPI (Message Passing Interface)** for distributing tasks across multiple processes and **thread-based parallelism** within each process.

Parallelization Strategy

1. Distributed Data Processing with MPI

- **Goal**: Split the workload of classifying test points across multiple processes to speed up computations.
- **Method**: Using MPI, the test data (`X_test`) is divided among the available MPI processes. Each process is responsible for predicting the labels of a subset of test points.
- **Implementation**:
  - The test dataset `X_test` is split across all MPI processes using: `local_X_test = np.array_split(X_test, size)[rank]`
  - Here, `size` is the total number of MPI processes, and `rank` identifies each process's unique ID. This command ensures each process works on only a fraction of `X_test`.

2. Multi-Threaded Distance Calculations Within Each Process

- **Goal**: Speed up the distance calculation between each test point and all training points by parallelizing this work within each process.
- **Method**: Within each MPI process, `ThreadPoolExecutor` is used to divide the training data (`X_train`) across multiple threads, so that distance calculations for each test point are performed concurrently.
- **Implementation**:
  - The training data is split into smaller subsets, one for each thread: `training_subsets = np.array_split(knn.X_train, num_threads)`
  - Each thread then calculates distances in parallel using `ThreadPoolExecutor`:

```
with ThreadPoolExecutor(max_workers=num_threads) as executor:
    distances_parts = executor.map(
        lambda subset: [knn.euclidean_distance(x, x_train) for x_train in subset],
        training_subsets
    )
```

  - Results from each thread are combined into a single array using:
    ```
    distances = np.concatenate(list(distances_parts))
    ```

3. Gathering Results with MPI

- **Goal**: Collect predictions from all MPI processes back to the root process for final verification and output.
- **Method**: Once each MPI process has completed its predictions on its subset of `X_test`, `comm.gather` is used to send these results back to the root process.
- **Implementation**:
  - `comm.gather` gathers predictions from all MPI processes:
    ```
    all_predictions = comm.gather(local_predictions, root=0)
    ```
  - On the root process (`rank 0`), all results are concatenated into a single array to match the sequential output:
    ```
    if rank == 0:
        return np.concatenate(all_predictions)
    ```

Important MPI Commands

- `MPI.COMM_WORLD`: Creates a communicator that includes all the MPI processes in this job.
- **`comm.Get_rank()`**: Returns the unique rank (ID) of the process within the communicator, used to identify each process's specific role.
- **`comm.Get_size()`**: Returns the total number of MPI processes in the communicator, which helps divide the workload.
- **`comm.gather(data, root=0)`**: Gathers data from all processes to the specified root process. In this case, it gathers predictions from all processes to the root (`rank 0`).

Summary of Benefits

This approach combines **distributed processing** (via MPI) with **multi-threading** to make efficient use of both multiple nodes and CPU cores within each node. The strategy effectively reduces computation time by splitting the test data across processes and parallelizing intensive calculations within each process. This hybrid approach is scalable and well-suited for large datasets and high-performance computing environments.

Benchmarking Overview for `benchmarking.py`

The `benchmarking.py` script compares the performance of sequential and parallel K-Nearest Neighbors (KNN) classifications over **30 runs**, measuring both **real time** (wall-clock) and **CPU time**. The parallelization uses **MPI** for distributing tasks across processes and **threading** within each process for faster distance calculations.

Benchmarking Strategy

1. **Command-Line Arguments**:
    - `--num_runs`: Specifies the number of benchmarking runs (set to 30 for HPC).
    - `--num_threads`: Sets the number of threads each MPI task uses.
2. **Functions for Parallel Prediction**:
    - `predict_single_test_point`: Uses threads within each MPI process to calculate distances to training points in parallel.
    - `parallel_predict`: Distributes test points across MPI processes and uses threading for intra-process parallelism. Gathers predictions from all processes to the root.
3. **Benchmarking Logic**:
    - **Sequential Benchmarking** (on root process only): Measures sequential prediction times for each run, records real and CPU times, and verifies correctness.
    - **Parallel Benchmarking** (all MPI processes): Each process predicts its assigned test points. Real and CPU times are measured on the root process, which verifies correctness and calculates speed-up.
4. **Result Calculation**:
    - On the root process, the script calculates:
        - **Average and standard deviation** of real and CPU times for both sequential and parallel predictions.
        - **Speed-up** by comparing average sequential and parallel real times.
    - Results are saved to `benchmark_report.txt`.

SLURM Batch Script (`run_knn_benchmarking.sh`)

The SLURM script configures the HPC environment:

- **Resource Allocation**:
    - `--ntasks=16`: 16 MPI tasks (processes).
    - `--cpus-per-task=8`: 8 CPU cores per task (128 CPUs in total).
- **Execution**:
    - `mpiexec -n 16 python benchmarking.py --num_runs 30 --num_threads 8`: Runs the benchmarking script with 16 MPI tasks, each using 8 threads.

This setup enables efficient parallel benchmarking to assess the time and resource savings from parallelizing the KNN classifier.

Benchmarking Results Summary

Correctness

- Both **sequential** and **parallel** versions produced **743 correct predictions**, confirming that parallelization did not impact accuracy.

Execution Times

- **Sequential Times**:
  - **Average Real Time**: 692.86 seconds
  - **Average CPU Time**: 694.25 seconds
  - **Standard Deviation (Real)**: 2.94 seconds
  - **Standard Deviation (CPU)**: 2.90 seconds
- **Parallel Times**:
  - **Average Real Time**: 44.03 seconds
  - **Average CPU Time**: 43.91 seconds
  - **Standard Deviation (Real)**: 0.24 seconds
  - **Standard Deviation (CPU)**: 0.18 seconds

Speed-Up

- **Real Time Speed-Up**: **15.74x**
  - This speed-up is close to the ideal factor of 16, which aligns with the parallel resources used:
    - **16 MPI tasks** (processes) each handling part of the workload.
    - **8 threads per process** handle distance calculations concurrently, fully utilizing available CPUs.
  - This combined parallelism effectively distributes work across 128 CPUs, achieving near-linear speed-up by reducing time spent on distance calculations and predictions.