PART 1

4. Set timers to calculate da4 with different “chunks”: (17 pts)

* + set the size of “chunks” to (1000, 1000).    (2 pts)
  + set the size of “chunks” to (3000, 2000).    (2 pts)
  + set the size of “chunks” to (300, 200)         (2 pts)
  + Plot a bin graph of the time with above different “chunks” (5 pts); and analyze the reason.    (6 pts)

***A graph of a graph of a graph

AI-generated content may be incorrect.***

***Analysis:***

Larger Chunk result in few task computation cycle and smaller chunks result in more task computation cycles. Each task has overhead such as scheduling, communicating and synchronization.

Larger Chunk size results to core being idle. Small Chunks result in more parallel Behavior but slows down due to above mentioned overhead.

Chunk Size of (3000, 2000) turns out to be the sweet spot in this use case since it executes the fastest.

PART 2

2. Set a timer to train the xgboost model by setting the n\_jobs to get the calculation time on CPU=1, 2, 4, 8; and plot the time varies on the number of CPUs. (8 pts)

A graph with a line

AI-generated content may be incorrect.

***Analysis***: The model training time keeps on reducing significantly as we keep adding more CPUs. The cutting down of time in half shows strong parallel efficiency. However, after 4CPUs to 8CPUs the improvement is smaller.

4. Use dask xgboost on 4 CPUs (5 pts) and dask distributed scheduler (5 pts) with n\_workers=4, threads\_per\_worker=1 to train the model and get the calculation time. Compare (3 pts) the calculation time of dask xgboost with the time of xgboost model on 4 CPUs. (13 pts)

A graph of blue squares

AI-generated content may be incorrect.

Analysis:

Dask XGBoost significantly outperforms the standard XGBoost in training time while being on same number of cores (cores=4).

Xgboost takes about 2.6 seconds while the dask-xgboost takes about 1 sec which is 2.5x speedup.

PART 3

4. You are required to take a screenshot of Dask Dashboard for running the step 3 with the 3 observed processes “Dask Process”, “Dask Graph”, “Dask Task Stream” in a same JupyterLab window, or use static Dask performance report. (Hint: since the dataset is small, the execution would be very fast. You may need to take a very quick screen capture).      (15 pts)

***Dashboard*** **(screenshot below)**

A screenshot of a computer

AI-generated content may be incorrect.

Let’s break down the sections below as required by the Question:

1) **Task Processing or Dask Process** (Task Processing/ CPU utilization) **(screenshot below)**:

A screenshot of a computer

AI-generated content may be incorrect.

Shows Tasks being processed by each worker i.e CPU utilization by each worker

Blue: processing tasks.

Green: worker has enough work to stay busy.

Red: Idle, doesn’t have enough work to stay busy.

**2) The Task Stream or Dask Task Stream** **(screenshot below)**

A chart with different colored squares

AI-generated content may be incorrect.

below shows the view of tasks across worker-threads.

Each color denotes a status:

Red – transferring data between workers,

Yellow – Disk I/O to collect local data.

Grey = serialization and deserialization of data.

Black = erred tasks.

White is execution time.

**3) Task graph or Dask Graph (screenshot below)**:

A screenshot of a computer

AI-generated content may be incorrect.

Blue color – task released

Red – stored in memory, waiting to be released

Grey – Waiting for worker to be assigned

Green – Processing or computing

The lines describe the transition of status between nodes.

4) ***Progress Bar* (screenshot below)**: it shows the progress of individual task.

A colorful lines on a white background

AI-generated content may be incorrect.

Dark green represents tasks are completed and released from memory

Green: tasks are completed and are in memory

Grey: tasks are ready to run

Lighter grey: tasks are queued, ready to run, not assigned any workers