Fast-Scalable-and-Easy-Machine-Learning-With-DAAL4PY

(Data Analytics Acceleration Library by Intel)

Project Report

Author

Abhishek Maheshwarappa

Kartik Kumar

Contents

1. Introduction
    a. Daal4py
        i. Background
        ii. Motivation
        iii. Goal
2. Methodology
    a. Daal4py methodology
3. Description of Datasets
    a. Telecom Churn Dataset
    b. Iowa Liquor Sales
    c. Crimes in Chicago
4. Result and Analysis
5. Guide to use the environment and Code

Introduction

The aim of this project is to work on and understand the effect of distributed and multiprocessing on computationally heavy jobs. We perform two case studies - Daal4py and SHAP. Daal4py was created to give data scientists the easiest way to utilize Intel(R) one API Data Analytics Library powerful machine learning building blocks directly in a high-productivity manner. Internally, Daal4py uses Intel(R) one API Data Analytics Library to deliver the best performance. We benchmark the performance with large datasets to see how scalable Daal4py is.

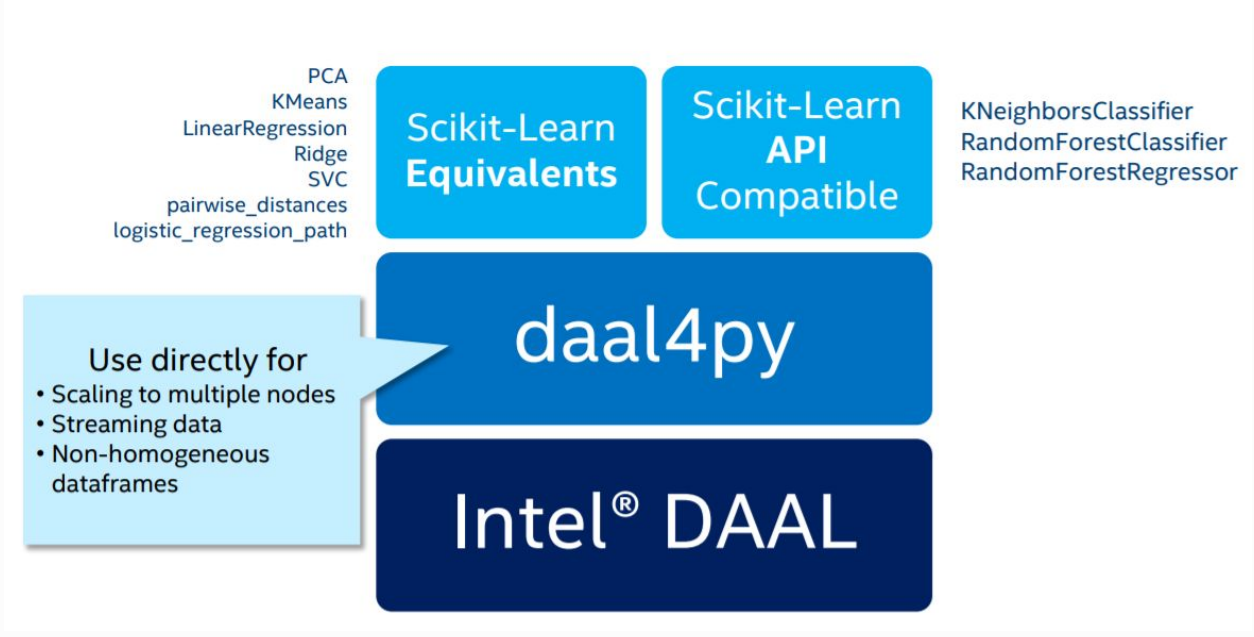
The benchmarking of Daal4py is a part of research under AI Skunkworks, in collaboration with Discovery (Research Computing at Northeastern University). The broad aim of the research is to benchmark PyDAAL(a.k.a. Daal4py) by Intel and compare it with other existing libraries for performance, scalability, and ease of use. This research is undertaken by Abhishek Maheshwarappa and Kartik Kumar.
.

Daal4py

Background

Daal4py makes machine learning algorithms in Python lightning-fast and easy to use. It provides highly configurable machine learning kernels, some of which support streaming input data and/or can be easily and efficiently scaled out to clusters of workstations. Internally it uses Intel(R) one API Data Analytics Library to deliver the best performance.

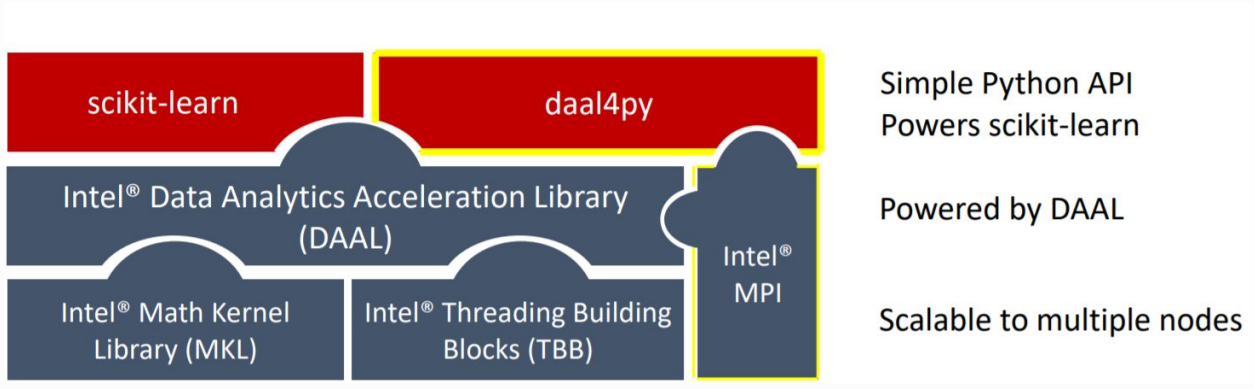
Daal4py was created to utilize Intel(R) one API Data Analytics Library powerful machine learning building blocks directly in a high-productivity manner. A simplified API gives high-level abstractions to the user with minimal boilerplate, allowing for quick to write and easy to maintain code when utilizing Jupyter Notebooks. For scaling capabilities, Daal4py also provides the ability to do distributed machine learning, giving a quick way to scale out. Its streaming mode provides a flexible mechanism for processing large amounts of data and/or non-contiguous input data.



The package uses Jinja templates to generate Cython-wrapped oneDAL C++ headers, with Cython as a bridge between the generated oneDAL code and the Python layer. This design allows for quicker development cycles and acts as a reference design to those looking to tailor their build of daal4py. Cython also allows for good Python behavior, both for compatibility to different frameworks and for pickling and serialization.

Motivation

Daal4py claims to be a faster and efficient way to run machine learning models, be it serial or parallel. The multiprocessing in Daal4py is called distributed computing (single Program Multiple Data - SPMD), which is claimed to be very fast compared to the native Scikit-learn algorithm which uses job lib for parallelism.



Goal

The goal is to test serial and parallel execution of computationally heavy jobs within the Daal4py framework. We want to test serial and parallel performance across multiple datasets and various algorithms.

Methodology

Daal4py methodology

There are 3 types of data processing in Daal4py. They are: -

1. Batch Processing
2. Stream Processing
3. Distributed Processing

Batch Processing

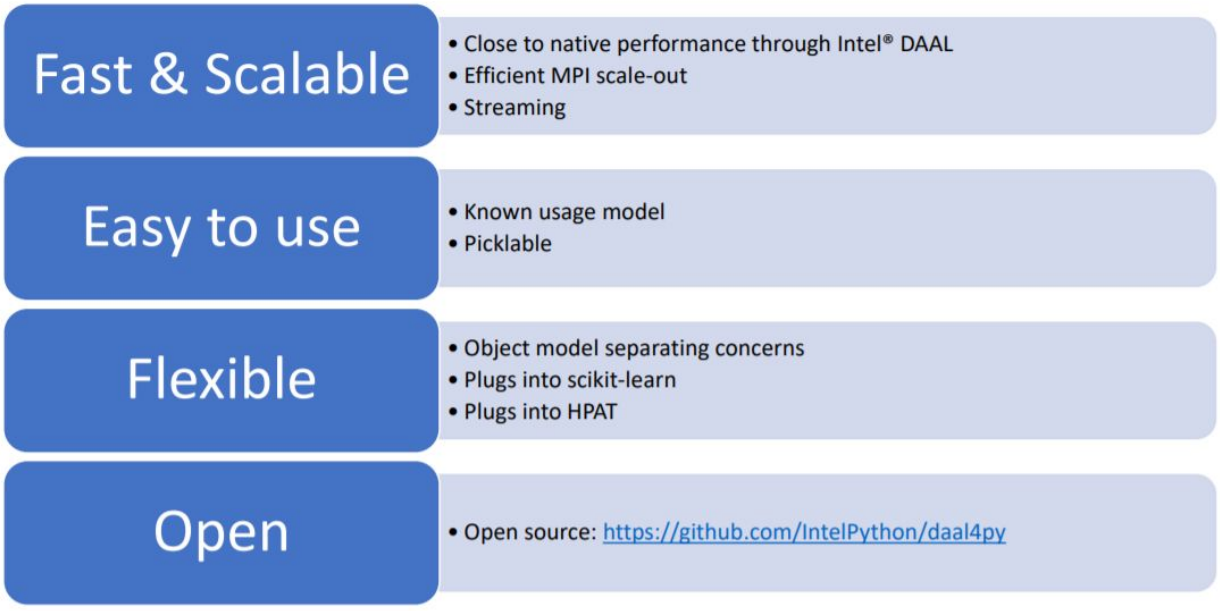
For small quantities of data, the input data can be inputted all at once using batch processing mode. This is similar to serial processing.

Stream Processing

For large quantities of data, it may be unfeasible to provide all input data at once unlike with batch processing. This might be because the data resides in multiple files and merging it is too costly to deem feasible. In other cases, the dataset may be too large to load completely into memory. The data being processed may also be an actual stream. Daal4py's streaming mode allows one to process these types of data in a quick and easy manner.

Distributed Processing

Daal4py operates in Single Program Multiple Data (SPMD) style, which means your program is executed on several processes (e.g. similar to MPI). The use of MPI is not required for Daal4py's SPMD-mode to work- all necessary communication and synchronization happens through Daal4py. However, it is possible to use Daal4py and mpi4py in the same program.



Under the hood, Daal4py uses MPI to implement SPMD. With basic initializations and function calls, the library uses a transceiver that initializes MPI routines and manages the communications internally.

Only very minimal changes are needed to your Daal4py code to allow Daal4py to run on a cluster of workstations.

1) Initialize the distribution engine using ***daalinit()***
2) Add the distribution parameter to the algorithm construction:
   kmi = kmeans_init(10, method="plusPlusDense", distributed=True)
3) When calling the actual computation, each process expects an input file or input array/DataFrame. The program needs to tell each process which file/array/DataFrame it should operate on. Like with other SPMD programs this is usually done conditionally on the process id/rank - ***daal4py.my_procid()***. Assume we have one file for each process, all having the same prefix ‘file’ and being suffixed by a number. The code could then look like this:

   result = kmi.compute("file{}.csv", daal4py.my_procid())
4) The result of the computation will now be available on all processes. Finally, stop the distribution engine using ***daalfini()***

Like all modern python analytics libraries, Daal4py contains a wrapper class for the low level c and c++ code. This code helps understand how the mpi initializations are called and how the objects communicate with each other. Here is a code snippet from the wrapper_gen.py which shows that the daalinit() function is actually using nthreads = -1. This is very similar to n_jobs=-1 provided by Scikit-learn algorithms which essentially mean that our code uses all CPUs available.

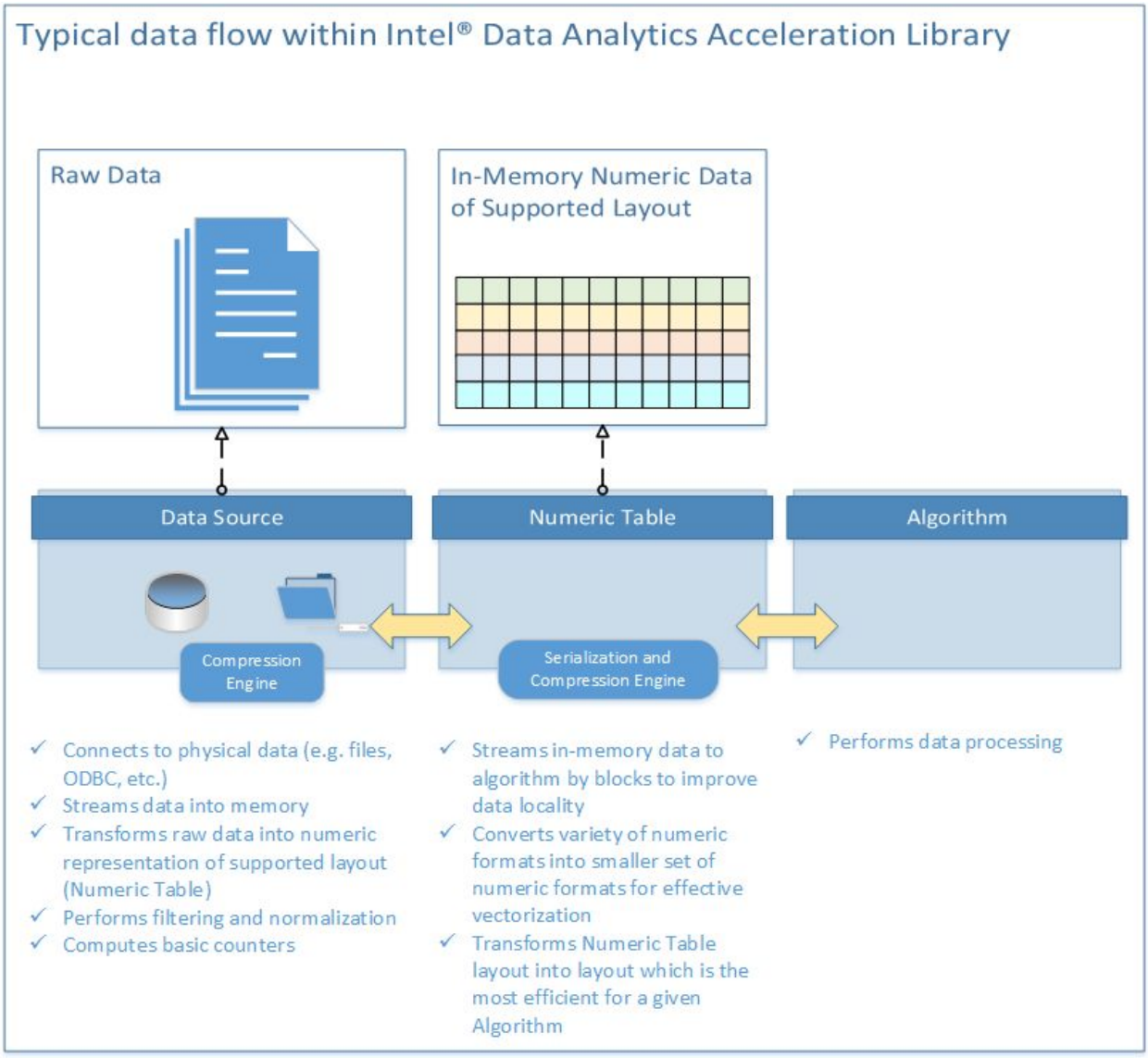
```
def daalinit(nthreads = -1):
    c_daalinit(nthreads)

def daalfini():
    c_daalfini()

def num_threads():
    return c_num_threads()

def num_procs():
    return c_num_procs()

def my_procid():
    return c_my_procid()
```

Distributed Processing - Input Data

Effective data management is among the key constituents of the performance of a data analytics application. For Intel® Data Analytics Acceleration Library (Intel® DAAL), data management requires effectively performing the following operations:

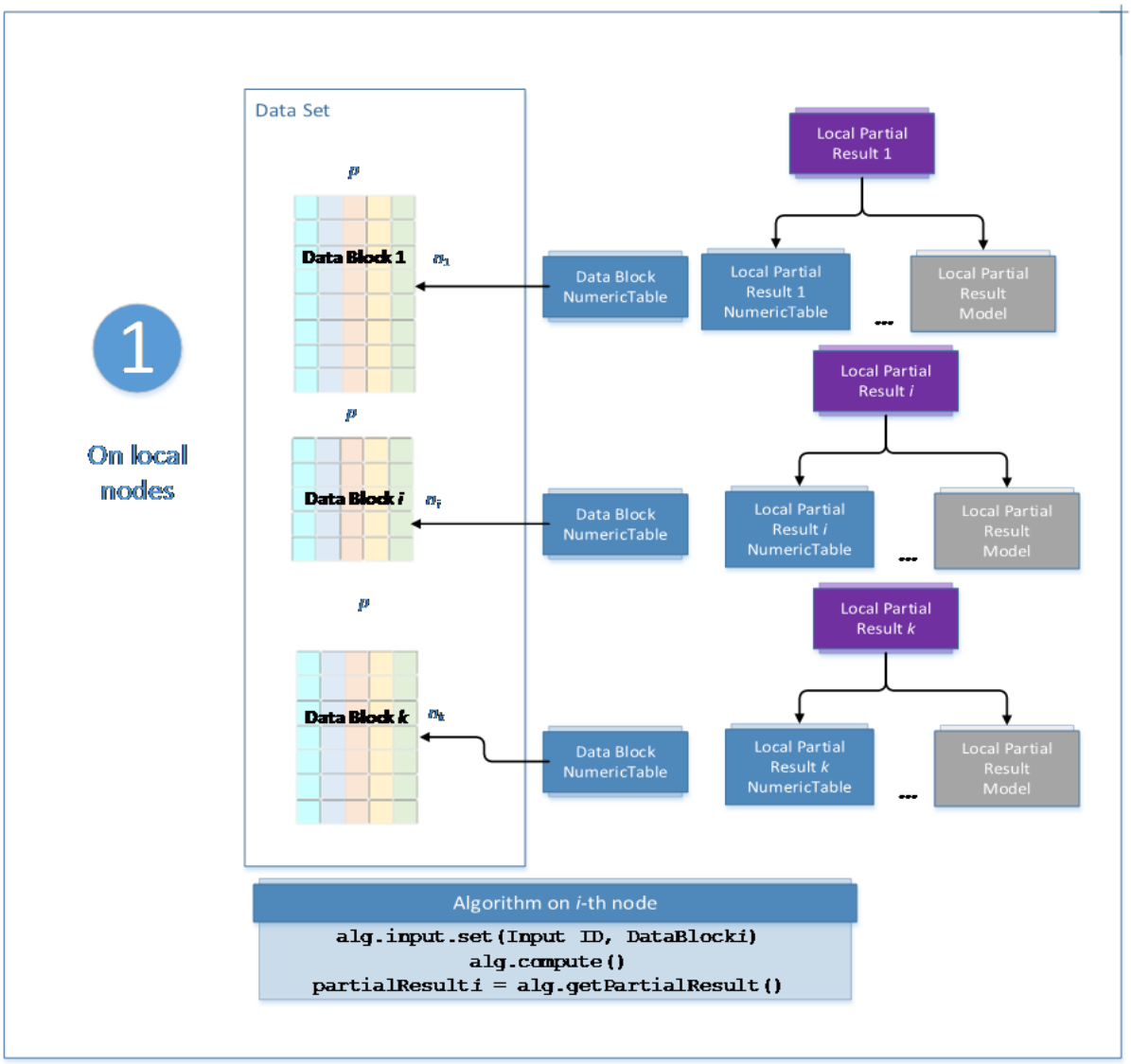
1. Raw data acquisition, filtering, and normalization with data source interfaces.
2. Conversion of the data to a numeric representation for numeric tables.
3. Data streaming from a numeric table to an algorithm

The below figure shows the typical data flow using Intel DAAL.



Distributed Processing - Algorithms

In the distributed processing mode, the Intel DAAL algorithm operates on a data set distributed across several devices (compute nodes), then produces partial results on each node, which are finally merged into the final result on the master node.



Steps involved in Multiprocessing or Distributed processing:

Local Nodes

In this step, the algorithm performs training on the local nodes with their own Data Block and a local partial model is being generated

Master Node

In this step, all the models generated at the local node combine to form a full model. The prediction of the result has to be a node at this node.

Description of Datasets

Telecom Churn Dataset

Maintaining current customers is very important as acquiring new customers is very expensive compared to maintaining current customers. So to understand what rate the customers are leaving Churn is calculated. The dataset contains the customer churn which is calculated by the number of customers who leave the company during a given period. The target variable in the dataset is 'Churn'. There may be many reasons for customer churn like bad onboarding, poor customer service, less engagement, and others.

Data Set Characteristics: Classification **-** Bivariate

Number of Instances: 6499

Target - Churn

Download Link

https://go.squarkai.com/squark_predict_churn_machine_learning_datasets.zip

Iowa Liquor Sales

The Iowa Department of Commerce requires that every store that sells alcohol in bottled form for off-the-premises consumption must hold a class "E" liquor license (an arrangement typical of most of the state alcohol regulatory bodies). All alcoholic sales made by stores registered thusly with the Iowa Department of Commerce are logged in the Commerce department system, which is in turn published as open data by the State of Iowa.

This dataset contains information on the name, kind, price, quantity, and location of the sale of sales of individual containers or packages of containers of alcoholic beverages.

Data Set Characteristics: Regression

Number of Observations: 12591077

Target - Bottles Sold

Download Link - https://www.kaggle.com/residentmario/iowa-liquor-sales

Crimes in Chicago

This dataset reflects reported incidents of crime (with the exception of murders where data exists for each victim) that occurred in the City of Chicago from 2001 to the present, minus the most recent seven days. Data is extracted from the Chicago Police Department's CLEAR (Citizen Law Enforcement Analysis and Reporting) system. In order to protect the privacy of crime victims, addresses are shown at the block level only and specific locations are not identified.

Data Set Characteristics: Classification - Bivariate
Number of Observations: 2688710

Target - Arrest

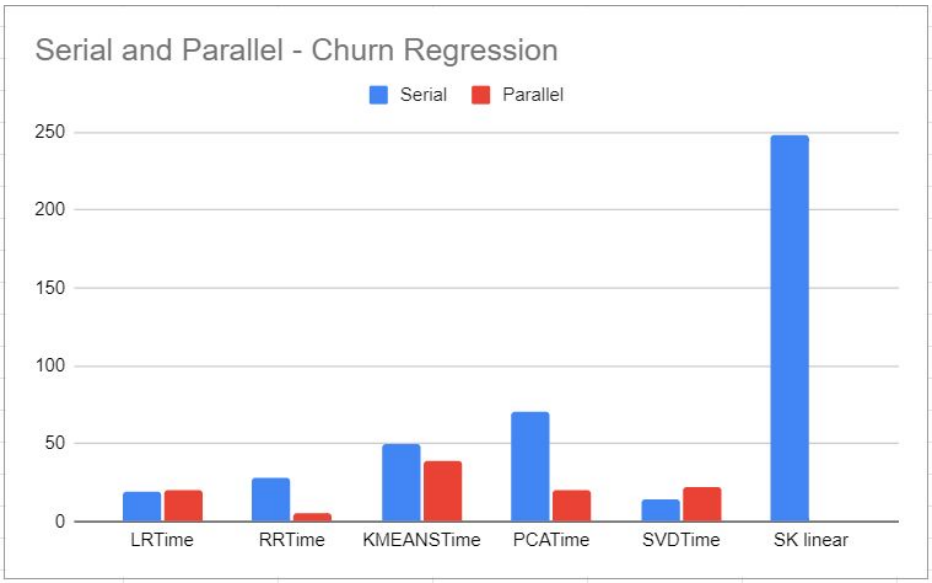
Download Link - https://www.kaggle.com/currie32/crimes-in-chicago

Result and Analysis

All-time durations shown in graphs are in milliseconds (ms)

Algorithms and abbreviations:
LR - Linear regression (daal4py) , RR - Ridge regression (daal4py), Kmeans - KMeans Clustering (daal4py), NB - Naive Bayes (daal4py), PCA Principal Component Analysis(daal4py), SVD - Singular Value Decomposition (daal4py), SK linear - Scikit -learn Linear Regression (sklearn)

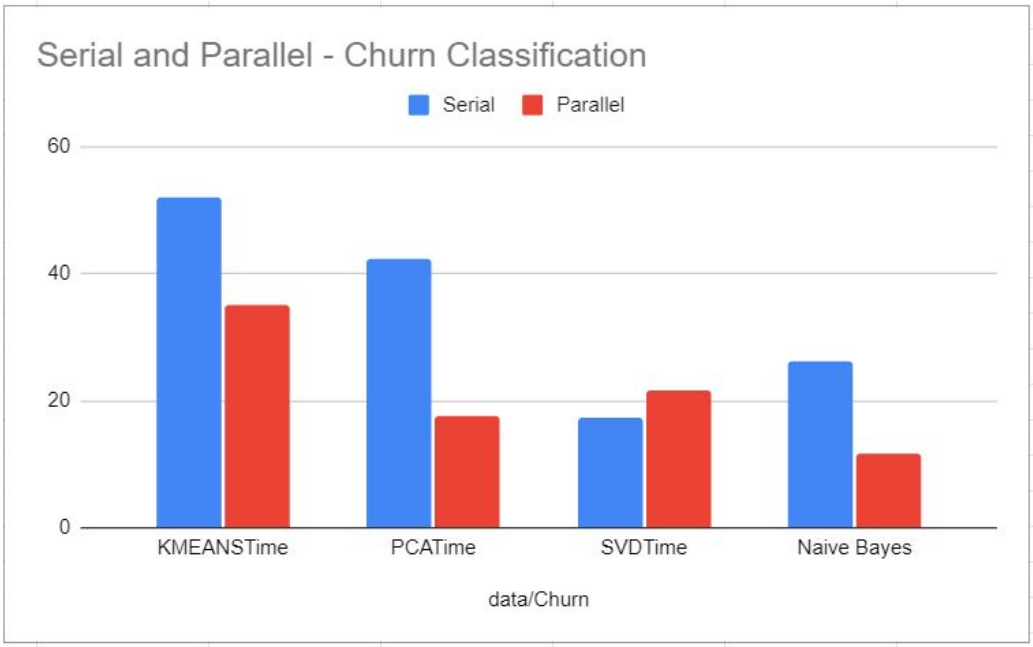
1. Churn Regression



As seen in the graph above, daal4py algorithms are much faster than Scikit-learn algorithms. Serial vs Parallel yields mixed results across algorithms but the overall time taken is very less

SLURM Command:
srun --partition=express --nodes=1 --cpus-per-task=1 --pty --mem=100gb --time=01:00:00 /bin/bash

2. Churn Classification

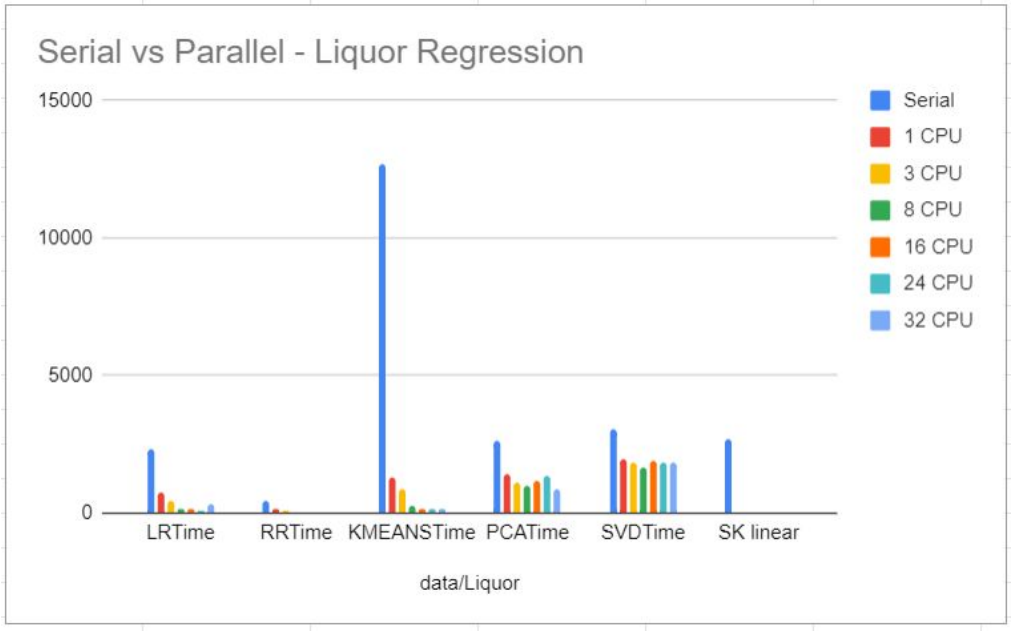


The graph above shows that parallel execution is almost 2x faster than serial across algorithms with a single node and single CPU

SLURM Command:

srun --partition=express --nodes=1 --cpus-per-task=1 --pty --mem=100gb --time=01:00:00 /bin/bash

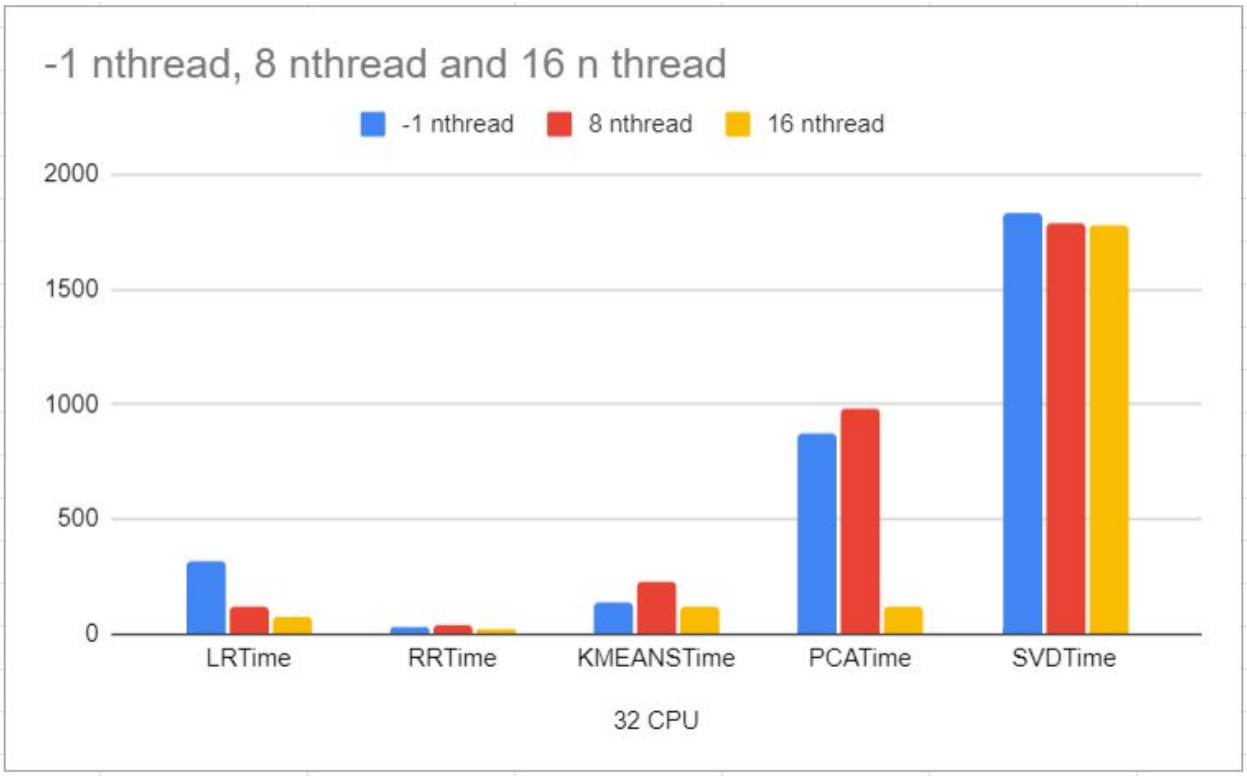
3. Liquor Regression
    a) Serial vs Parallel (CPU count variation)



The above graph shows how an increase in CPUs reduces algorithm computation speed. The parallel execution is much faster than serial and sklearn. A general trend is that time reduces with an increase in the number of CPUs but in some cases after reaching optimal setting, the time may vary by small amounts.

SLURM Command:
srun --partition=express --nodes=1 --cpus-per-task=**[1,3,8,16,32]** --pty --mem=100gb --time=01:00:00 /bin/bash

b) Variations with nthreads parameter



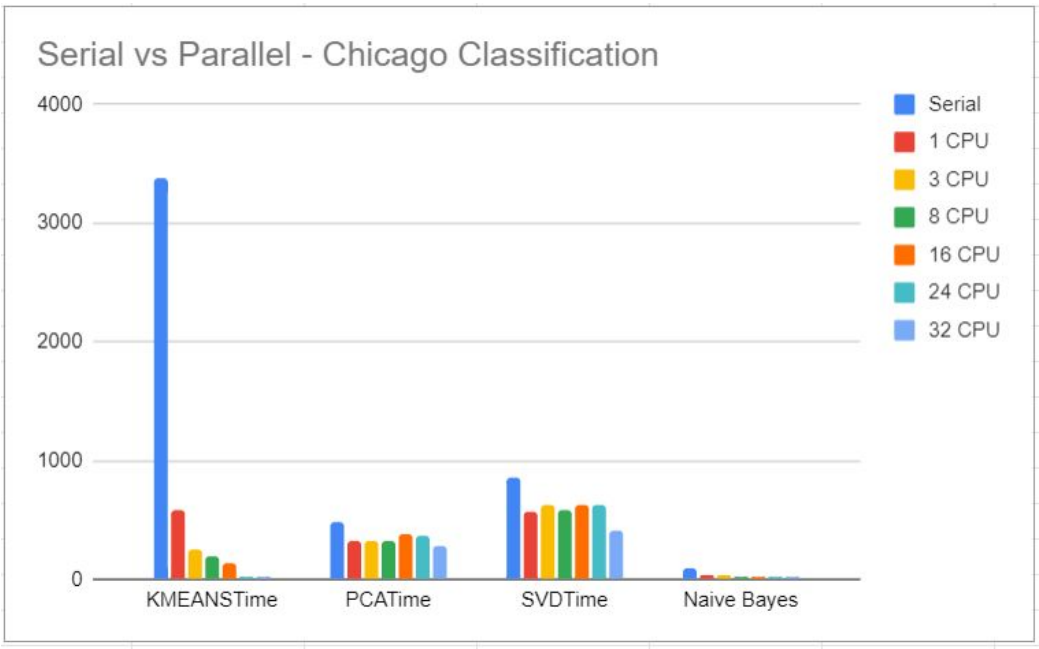
The nthreads parameter has a default value of -1 in the daalinit() initialization. This means that the parallelization uses all CPUs available and executes the process. We tried varying this parameter and seeing the results.

As seen in the graph above, the performance varies with the change in the value of this parameter. With 32 CPUs available, nthreads=8 uses only 8 and some algorithms yield better performance with this setting.

Interestingly, the nthreads = 8 and nthreads=16 performance is better than that when the Slurm command had requested for 8 and 16 CPUs respectively, across all algorithms.

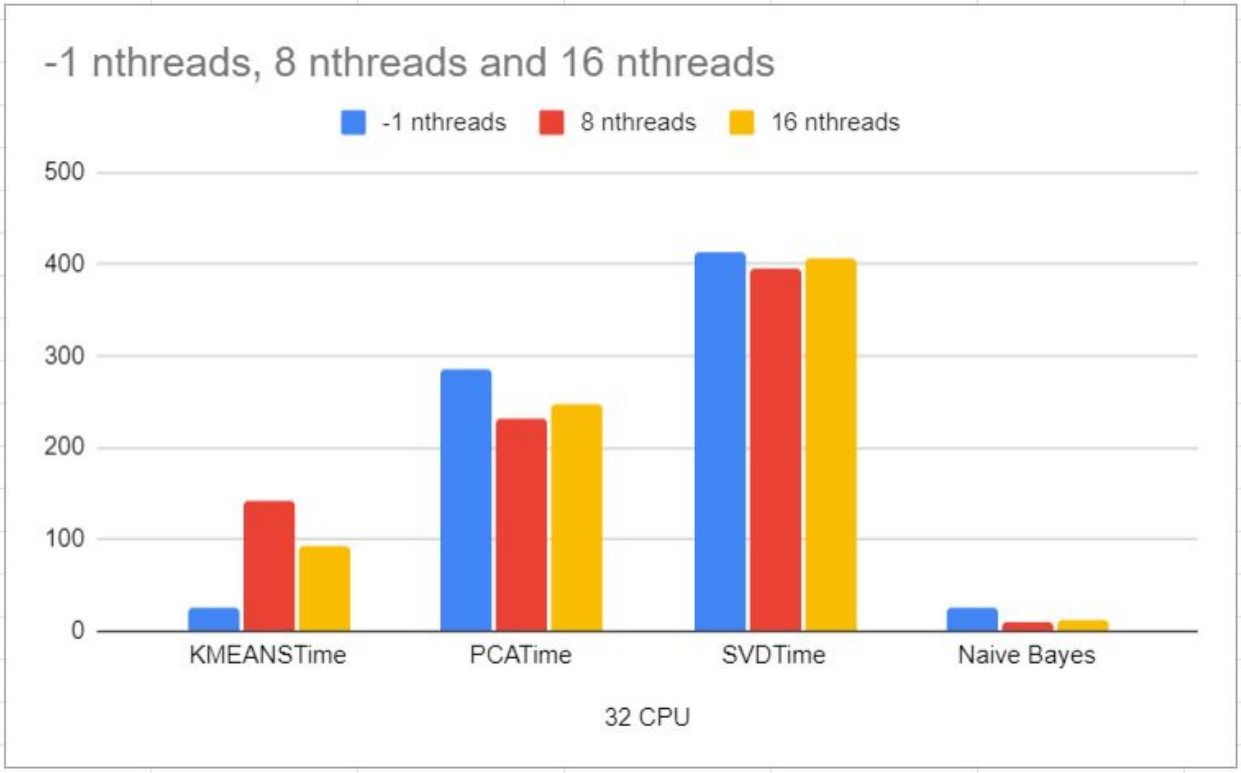
SLURM Command:
srun --partition=express --nodes=1 --cpus-per-task=32 --pty --mem=100gb --time=01:00:00 /bin/bash

4. Chicago - Classification
    a) Serial vs Parallel (CPU count variation)



As seen in the plot above, the parallel execution is faster than the serial execution across all classification algorithms. The increase in CPUs improves the time performance.

b) Variations with nthreads parameter



The change in the value of nthreads has a significant impact on performance when 32 CPUs are available. For every algorithm, there is a unique value of the parameter that yields the best performance.

As in the Liquor Regression case, the nthreads = 8 and 16 yields better performance than when the code ran with 8 and 16 CPUs available to it.

This is an interesting find as the nthreads parameter is not shown anywhere in the documentation of dal4py. We had to dive into the source code on GitHub to find it. :)

Guide to use the environment and Code

Steps to setup environment to use Daal4py

1. We have a daal4py.yml file on the github repo and in the zip folder, locate it.

2. Go to anaconda or miniconda installed on the cluster and create the virtual environment.
3. While creating the environment use yml file provided
    a. Create the environment from the `environment.yml` file:
    `conda env create -f environment.yml`
    b. Activate the new environment: `conda activate myenv`

c. Verify that the new environment was installed correctly:

```
conda env list
```

Steps to run the code

1. Once the environment is set up it will allow you to run Daal4py
2. Navigate to the directory containing main.py file
3. There few datasets in the data folder and any other tabular data can be added
4. To run use the command python main.py

Once we run the user flow looks like this

```
(daal4py) [maheshwarappa.a@c0173 Benchmark-daal4py]$ python main.py



************************************************************
-----------------------Data-----------------------


sales.csv
nfip-flood-policies.zip
insurance.csv
new.py
food.csv
Churn.csv
Chicago_Crimes.csv
Liquor.csv
clv.csv
diabetes.csv
epa_hap_daily_summary.csv.zip




Which Data to train?
Churn
enter num of threads
-1

 The columns present are

 Index(['CustomerID', 'Gender', 'Senior Citizen', 'Partner', 'Dependents',
       'Tenure', 'Phone Service', 'Multiple Lines', 'Internet Service',
       'Online Security', 'Online Backup', 'Device Protection', 'Tech Support',
       'Streaming TV', 'Streaming Movies', 'Contract', 'Paperless Billing',
       'Payment Method', 'Monthly Charges', 'Total Charges', 'Churn'],
      dtype='object')

Choose the target coulum

Churn
you want classfication or not ?

Note - PCA, SVD and Kmeans run irrespective of the classification or not

Classification:
1 - True or 2 - False
1
Run options
1 - Serial

2 - Parallel

Want to run parallel or serial?
1
```

Main.py

Is the function which handles all the function

```
class mains():

    def __init__(self):
        # getting the current system time
        self.current_time = datetime.datetime.now().strftime("%Y-%m-%d__%H.%M")

        # declaring variables and data structures
        # latency dictionary to hold execution times of individual functions
        self.latency = dict()

        # metric dictionary

        self.metrics = dict()

        # removing any existing log files if present
        if os.path.exists(target_dir + '/main.log'):
            os.remove(target_dir + '/main.log')

        # get custom logger
        self.logger = self.get_loggers(target_dir)

    @staticmethod
    def get_loggers(temp_path):
        # name the logger as HPC-AI skunkworks
        logger = logging.getLogger("HPC-AI skunkworks")
        logger.setLevel(logging.INFO)
        # file where the custom logs needs to be handled
        f_hand = logging.FileHandler(temp_path + '/' + key+'.log')
        f_hand.setLevel(logging.INFO)  # level to set for logging the errors
        f_format = logging.Formatter('%(asctime)s : %(process)d : %(levelname)s : %(message)s',
                                     datefmt='%d-%b-%y %H:%M:%S')
        # format in which the logs needs to be written
        f_hand.setFormatter(f_format)  # setting the format of the logs
        # setting the logging handler with the above formatter specification
        logger.addHandler(f_hand)

        return logger

    def data_split(self, data):
        '''
        This funtion helps to generate the data
        required for multiprocessing
        '''
        self.logger.info(" The Data spliting process started")
```

Custom logger

This helps to track all the things happening in the entire script and log them to track any errors while debugging.

Latency

This is a dictionary used to track all the time taken by different functions in the script and will be saved as a json file for every run.

Metrics

This is the dictionary to save all the metrics of the trained model and will be saved as a json file for every run.

Create directory

WE create a new temporary directory for every run where our results for that particular run will be saved.

All the other script are called from this script

Serial script and parallel script contains the following Algorithms

1. Linear regression
2. Ridge Regression
3. PCA
4. SVD
5. Naive Bayes
6. K-Means

These scripts are all commented on for easy understanding.