Machine Learning Predictions for estimation of Job wait time on HPC (High Performance Computing) System

Deenadayalan Dasarathan

Master of Science in Data Science, Khoury College of Computer Sciences, Northeastern University, Boston, Massachusetts

dasarathan.d@northeastern.edu

MLPredictionsOnJobWaitTime

Abstract

Multiple processes or jobs may contend for shared resources in the HPC system environment, thus each job suffers a wait time ranging from few minutes to days, to get the requested resource for the execution. The job scheduler in the HPC system works as a black box, which does not provide any relationship between the queue time and requested resource. Thus, the user is unaware of the queue time that it is going to take before the start of execution of the job.

This regression problem can be solved by building a machine learning model to analyze the past job samples generated by the Discovery cluster users. The model predicts the queue wait time based on the resource the user requests. The estimation of this wait time helps the user to know the wait time before submitting the job to HPC.

Introduction

The HPC cluster typically hosts a variety of compute nodes and GPUs which are consumed by the users for their resource intensive tasks. This inevitably creates resource contention. Hence, most contemporary HPC systems use a scheduler system to provide fair use of the shared resources. On Discovery, a high-performance computing resource for the Northeastern University research computing, The SLURM (Simple Linux Utility for Resource Management) is an open-source resource management and job scheduling system manages the HPC cluster resources. When a job is submitted to the cluster, the SLURM schedules the job in the queue and allocates the requested resources for the prescribed time window. Often the submitted jobs undergo some wait time in the queue to get the requested resource (no of cpus, no of nodes, requested memory, gpu node, etc.). The queue wait time could be defined as

Queue time = Start time (when job starts its execution) – Submit time (when job submitted to SLURM)

In many cases there is trade-off between the queue time and requested resource. When more resources are requested, the job compute time will be faster, but the queue time will be longer to get the resource allocation. The knowledge on queue time is important for the users as it counts to the total time for execution of a job. Knowing this information before submitting the job, the user can adjust the required resource for computation to get the results sooner.

The estimates of queue wait time also benefits the HPC administrators as the SLURM scheduler can be modified to fit the specific workloads running on the cluster. In situations where many users are requesting the same resources, SLURM parameters can be tweaked to accommodate the job and thus reduce wait times.

Currently, it is possible to obtain an estimated queue wait time from the job scheduler only after the job has been submitted and executed. Additionally, the job scheduler works as a black box without showing a relationship between queue times and requested resources.

In this study, we propose a data-driven approach for predicting job wait time on HPC systems prior to submitting the job. Here, “data-driven” means that our approach actively observes and analyzes the job logs collected on Discovery cluster. Supervised machine learning algorithms are applied to predict the queue wait time for the job, which is dependent on several features like requested resource, resource availability, number of active jobs, number of pending jobs at that time.

Background

The users can submit their job to the HPC system using ‘sbatch’ or ‘srun’ command.

Below is the sample sbatch script to request 1 node with 128 CPUs with exclusive flag enabled i.e., entire memory in the compute node is available for the user.

```
#!/bin/bash
#SBATCH --partition=short
#SBATCH --job-name=model
#SBATCH --time=23:59:59
#SBATCH -N 1
```

```
#SBATCH -n 128
#SBATCH --exclusive
#SBATCH --output=%j.output
#SBATCH --error=%j.error
## load the anaconda package:
module load anaconda3/2021.05
## activate the environment
source activate conda3.8
## run the python script
python /home/d.dasarathan/project_sml/pycode/model.py
```

Below is the sample srun command,
srun --partition=short -N 1 -n 128 --pty --export=ALL --exclusive --time=23:59:59 /bin/bash

The SLURM daemon gets the job submit request and records this information in the SLURM database. The job detail is updated at different stages, when the SLURM assigns the requested resource until the job reaches its end of execution. This job information can be accessed via sacct command, which queries the SLURM DB.
I have analyzed job samples generated by Discovery cluster users to build a machine learning model that establishes a relationship between queue time and requested resources and predicts the queue time based on those resources.

Related Work

A paper titled 'Ensemble Prediction of Job Resources to Improve System Performance for SLURM-Based HPC Systems' published in PEARC'21 by Dr. Daniel Andresen and team. The objective of this paper is to model time (CPUTimeRaw) and memory (MaxRSS) as a function of requested parameters like Account, Time-Limit, ReqNodes, ReqMem, ReqCPUS and QOS. The paper has discussed six popular regression model from scikit-learn and Microsoft for the task. The models include Lasso Least Angle Regression, Linear Regression, Ridge Regression, Elastic Net regression, Classification and Regression Trees, Random Forest Regression and Microsoft LightGBM. The maximum score of 0.78 was achieved using Random Forest Regression model.

Project Description

Data Extraction

The job records of past are extracted from SLURM DB, by running a sacct command,
sacct --starttime=2019-10-01 --endtime=2020-03-25 -p -all
The result of above query is the pipe separated data which provides a lot of information about the jobs executed between the starttime and endtime. Below is the snippet of the raw dataset extracted from the database,

The dataset has more than 700,000 samples and 88 features. The raw data is converted into a csv file with delimiter '^'

```
$ sed 's/|/^/g' rawdata > rawdata.csv
```

The csv dataset is then serialized and converted into a pickle file, as it is much faster when compared to csv file.

```
import pandas as pd
import pickle
```

```
df_csv = pd.read_csv('/home/d.dasarathan/project_sml/rawdata.csv', delimiter='^')
```

```
df_csv.to_pickle('/home/d.dasarathan/project_sml/rawdata.pkl')
```

```
path ='/home/d.dasarathan/project_sml/100Kdataset/df_raw_600k.pkl'
df_raw_data=pd.read_pickle(path)
df_raw_data.head()
```

	Account	AdminComment	AllocCPUS	AllocGRES	AllocNodes	AllocTRES	AssocID	AveCPU	AveCPUFreq	AveDiskRead
0	f.hung	NaN	28.0	NaN	1.0	cpu=28,mem=56000M,node=1,billing=28	2814	NaN	NaN	NaN
3	f.hung	NaN	28.0	NaN	1.0	cpu=28,mem=56000M,node=1,billing=28	2814	NaN	NaN	NaN
6	allshouse	NaN	8.0	NaN	0.0	NaN	2755	NaN	NaN	NaN
7	sioannidis	NaN	80.0	NaN	5.0	cpu=80,mem=160000M,node=5,billing=80	3177	NaN	NaN	NaN
11	j.dy	NaN	1.0	NaN	1.0	cpu=1,mem=16G,node=1,billing=1	3065	NaN	NaN	NaN

The dataset has 88 features, out of which many features are of least significance and some are null values.
The following are the features extrscted from sacct,
['Account', 'AdminComment', 'AllocCPUS', 'AllocGRES', 'AllocNodes','AllocTRES', 'AssocID', 'AveCPU',

'AveCPUFreq', 'AveDiskRead','AveDiskWrite', 'AvePages', 'AveRSS', 'AveVMSize', 'BlockID', 'Cluster','Comment', 'ConsumedEnergy','ConsumedEnergyRaw', 'CPUTime','CPUTimeRAW', 'DerivedExitCode', 'Elapsed', 'ElapsedRaw', 'Eligible','End', 'ExitCode', 'GID', 'Group', 'JobID', 'JobIDRaw', 'JobName','Layout', 'MaxDiskRead', 'MaxDiskReadNode', 'MaxDiskReadTask','MaxDiskWrite', 'MaxDiskWriteNode', 'MaxDiskWriteTask', 'MaxPages','MaxPagesNode', 'MaxPagesTask', 'MaxRSS', 'MaxRSSNode', 'MaxRSSTask','MaxVMSize', 'MaxVMSizeNode', 'MaxVMSizeTask', 'McsLabel', 'MinCPU','MinCPUNode', 'MinCPUTask', 'NCPUS', 'NNodes', 'NodeList', 'NTasks','Priority', 'Partition', 'QOS', 'QOSRAW', 'ReqCPUFreq', 'ReqCPUFreqMin', 'ReqCPUFreqMax', 'ReqCPUFreqGov', 'ReqCPUS', 'ReqGRES', 'ReqMem', 'ReqNodes', 'ReqTRES', 'Reservation', 'ReservationId', 'Reserved', 'ResvCPU', 'ResvCPURAW', 'Start', 'State', 'Submit', 'Suspended', 'SystemCPU', 'Timelimit', 'TotalCPU', 'UID', 'User', 'UserCPU', 'WCKey', 'WCKeyID', 'WorkDir', 'Unnamed: 87']

The data set has the features captured from the sbatch or srun and several other important features are generated by SLURM based on the user past usage, user association to PI account, allocated memory and cpu core and time limit. The dataset may contain white spaces and junk samples, which removed manually before serializing it into a pickle file.

Data Treatment and Filtration

The data cleaning step involves a variety of task. The following job samples are removed from dataset,

- Jobs that never started i.e., start time = Nan
- Jobs that have start time before submit time.
- Jobs that have end time in the year 2021.

The data with value ‘unknown’ are replaced by NaN. The string values in the timestamp features are removed and the datatype is changed to pandas datatime type.

Feature Engineering

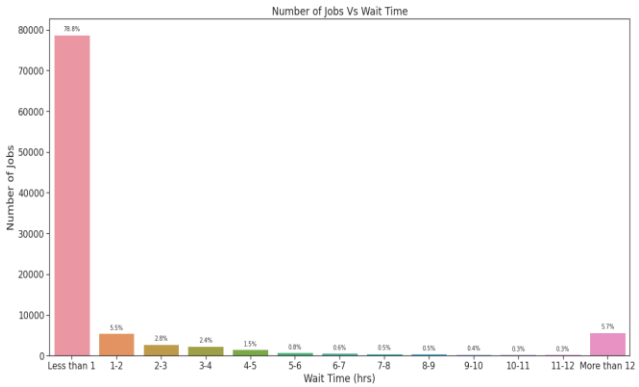
The target label is the queue time which is derived from features ‘Submit’ and ‘Start’, and its then converted into hours
waitTime = Start – Submit
The active or the running job count is an important feature which is calculated for each sample by comparing the Submit time with Start and End time of other samples. This feature calculation involves a lot of time, which is accelerated by using Swifter package with Pandas dataframe.
The ReqCPUS and TimeLimit features are combined into single feature called CoreHrs.
CoreHrs = TimeLimit in hrs * ReqCPUS
The ReqTotMem feature is standardized by converting the memory values in MB and TB into GB.
The exclusive flag is set when all the CPUS in the node allocated for the job.
QOD and QOY are derived from the submit time feature.
QOD as q1,q2,q3 or q4 based on the submit time hour.
QOY as q1,q2,q3 or q4 based on the submit time month.

Feature Name	Type	Description
ReqNodes	Int64	Requested number of nodes
Priority	Float64	Slurm computed value for each user
Partition	Object	Group of nodes
Corehrs	Float64	Timelimit * ReqCPUS
Req_totalMem	Float64	Derived from ReqMem, ReqCPUS and ReqNodes
Exclusice	Int64	Derived from NCPUS and ReqCPUS
Activejbcount	Int64	Derived from Submit, Start and End time
QOD	Object	Derived from submit time
QOY	Object	Derived from submit time

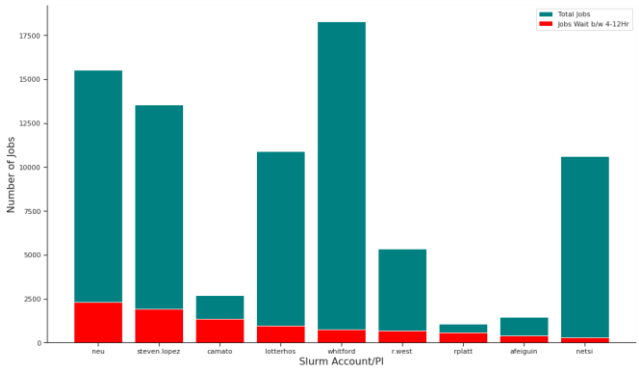
The dataset with above mentioned 9 features and target label is stored in a separate dataset for next phase of the ML lifecycle.

EAD

As part of explanatory data analysis, several plots are made to understand the queue time.



About 79% of the jobs in the entire dataset had a queue time less than an hour. The remaining 21% of the jobs suffered longer wait time for the resource allocation. This project would help the power users to know the queue time before submitting the job to HPC system. With the knowledge on queue wait time the user can modify the resource parameters in SBATCH or SRUN to get quicker execution time.



The above bar chart shows the relationship between the SLURM account to which the user is associated with and queue time. The chart indicates that there are more chances for the jobs submitted by users of common PI or SLURM account to have longer queue time than jobs submitted by users of dedicated PI accounts.

Train-Test Split

The scikit-learn's Train_Test_Split module is used to split the feature engineered dataset into train and test samples, which are then used to train and evaluate the model. The split is random with ratio 80% train and 20% test.

Modeling

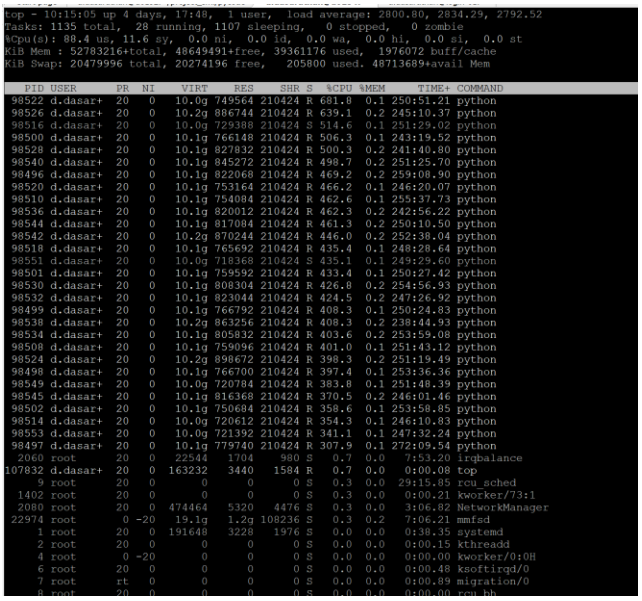
Random Forest Regressor

For this regression problem (predicting wait time) ensemble models are best candidates because of the bootstrapping and bagging techniques. I have used scikit-learn's Random Forest Regressior model to train and predict the wait time.

Initially I have trained a Random Forest Regression model with following hyper-parameters,

```
RandomForestRegressor(bootstrap=False,
                      max_depth=50,
                      min_samples_leaf=10,
                      min_samples_split=5,
                      n_estimators=400,
                      n_jobs=-1)
```

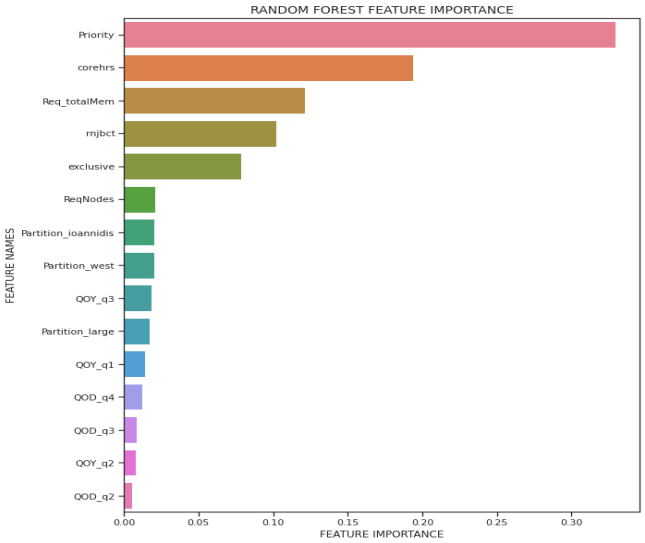
n_jobs=-1 enables the underlying decision trees to train and predict in parallel. This parallelization utilizes all the processors in the node.



```
top - 10:15:05 up 4 days, 17:48,  1 user,  load average: 2800.80, 2834.29, 2792.52
Tasks: 1135 total,  28 running, 1107 sleeping,   0 stopped,   0 zombie
%Cpu(s): 88.4 us, 11.6 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 52783216+total, 48649491+free, 39361176 used,  1976072 buff/cache
KiB Swap: 20479996 total, 20274196 free,   205800 used. 48713689+avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
98522 d.dasar+  20   0   10.0g 749564 210424 R 681.8  0.1 250:51.21 python
98526 d.dasar+  20   0   10.2g 886744 210424 R 639.1  0.2 245:10.37 python
98516 d.dasar+  20   0   10.0g 729388 210424 S 514.6  0.1 251:29.02 python
98500 d.dasar+  20   0   10.1g 766148 210424 R 506.3  0.1 243:19.52 python
98528 d.dasar+  20   0   10.1g 827832 210424 R 500.3  0.2 241:40.80 python
98540 d.dasar+  20   0   10.1g 845272 210424 R 498.7  0.2 251:25.70 python
98496 d.dasar+  20   0   10.1g 822068 210424 R 469.2  0.2 259:08.90 python
98520 d.dasar+  20   0   10.1g 753164 210424 R 466.2  0.1 246:20.07 python
98510 d.dasar+  20   0   10.1g 754084 210424 R 462.6  0.1 255:37.73 python
98536 d.dasar+  20   0   10.1g 820012 210424 R 462.3  0.2 242:56.22 python
98544 d.dasar+  20   0   10.1g 817084 210424 R 461.3  0.2 250:10.50 python
98542 d.dasar+  20   0   10.2g 870244 210424 R 446.0  0.2 252:38.04 python
98518 d.dasar+  20   0   10.1g 765692 210424 R 435.4  0.1 248:28.64 python
98551 d.dasar+  20   0   10.0g 718368 210424 S 435.1  0.1 249:29.60 python
98501 d.dasar+  20   0   10.1g 759592 210424 R 433.4  0.1 250:27.42 python
98530 d.dasar+  20   0   10.1g 808304 210424 R 426.8  0.2 254:56.93 python
98532 d.dasar+  20   0   10.1g 823044 210424 R 424.5  0.2 247:26.92 python
98499 d.dasar+  20   0   10.1g 766792 210424 R 408.3  0.1 250:24.83 python
98538 d.dasar+  20   0   10.2g 863256 210424 R 408.3  0.2 238:44.93 python
98534 d.dasar+  20   0   10.1g 805832 210424 R 403.6  0.2 253:59.08 python
98508 d.dasar+  20   0   10.1g 759096 210424 R 401.0  0.1 251:43.12 python
98524 d.dasar+  20   0   10.2g 898672 210424 R 398.3  0.2 251:19.49 python
98498 d.dasar+  20   0   10.1g 766700 210424 R 397.4  0.1 253:36.36 python
98549 d.dasar+  20   0   10.0g 720784 210424 R 383.8  0.1 251:48.39 python
98545 d.dasar+  20   0   10.1g 816368 210424 R 370.5  0.2 246:01.46 python
98502 d.dasar+  20   0   10.1g 750684 210424 R 358.6  0.1 253:58.85 python
98514 d.dasar+  20   0   10.0g 720612 210424 R 354.3  0.1 246:10.83 python
98553 d.dasar+  20   0   10.0g 721392 210424 R 341.1  0.1 247:32.24 python
98497 d.dasar+  20   0   10.1g 779740 210424 R 307.9  0.1 272:09.54 python
 2060 root      20   0   22544   1704    980 S   0.7  0.0   7:53.20 irqbalance
107832 d.dasar+ 20   0  163232   3440   1584 R   0.7  0.0   0:00.08 top
    9 root      20   0       0      0      0 S   0.3  0.0  29:15.85 rcu_sched
 1402 root      20   0       0      0      0 S   0.3  0.0   0:00.21 kworker/73:1
 2080 root      20   0  474464   5320   4476 S   0.3  0.0   3:06.82 NetworkManager
22974 root       0 -20   19.1g   1.2g 108236 S   0.3  0.2   7:06.21 mmfsd
    1 root      20   0  191648   3228   1976 S   0.0  0.0   0:38.35 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.15 kthreadd
    4 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    6 root      20   0       0      0      0 S   0.0  0.0   0:00.48 ksoftirqd/0
    7 root      rt   0       0      0      0 S   0.0  0.0   0:00.89 migration/0
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
```

The above snapshot shows the parallel processing of random forest regression model.



Above is the feature importance plot for the trained model.

Performance Metrics
R^2 Training Score: 0.90
R^2 Test Score: 0.88
Mean Absolute Error: 0.6 wait hours.

The performance of the trained Random Forest Model is good, but still the model can be tuned to perform better. The hyper-parameter tunning is done using scikit-learn's RandomizedSearchCV module.

Hyper-parameter Tuning using RandomizedSearchCV
Random Grid: {'
n_estimators': [50, 185, 321, 457, 592, 728, 864, 1000],
'max_features': ['auto', 'sqrt'],
'max_depth': [10, 15, 21, 27, 32, 38, 44, 50, None],
'min_samples_split': [2, 5, 10],
'min_samples_leaf': [1, 2, 4],
'bootstrap': [True, False]}

With cv=3 and n_iter=10
Fitting 3 folds for each of 10 candidates, totaling 30 fits.

Random Forest best Parameters: {
'n_estimators': 321,
'min_samples_split': 10,
'min_samples_leaf': 2,
'max_features': 'auto',
'max_depth': 21,
'bootstrap': True}

XGBRegressor

XGBoost stands for "Extreme Gradient Boosting" and it is an implementation of gradient boosting trees algorithm. The XGBoost is a popular supervised machine learning model with characteristics like computation speed, parallelization, and performance.

Initially I have trained the XGBRegrssor model with following parameters,
XGBRegressor(alpha=10,
base_score=None,
booster=None,
colsample_bylevel=None,
colsample_bynode=None,
colsample_bytree=0.3,
gamma=None,
gpu_id=None,
importance_type='gain',
interaction_constraints=None,
learning_rate=0.1,
max_delta_step=None,
max_depth=50,
min_child_weight=None,
missing=nan,
monotone_constraints=None,
n_estimators=400,
n_jobs=None,
num_parallel_tree=None,
random_state=None,
reg_alpha=None,
reg_lambda=None,
scale_pos_weight=None,
subsample=None, tree_method=None,
validate_parameters=None,
verbosity=None)

Performance Metrics
R^2 Training Score: 0.97
R^2 Test Score: 0.88
Mean Absolute Error: 0.58 wait hours.

Hyper-parameter Tuning using RandomizedSearchCV
With cv=3 and n_iter=10
Fitting 3 folds for each of 10 candidates, totaling 30 fits.

XGBoost best Parameters: {
'colsample_bytree': 0.7832290311184182,
'learning_rate': 0.02214485163452344,
'max_depth': 18,
'min_child_weight': 4,
'n_estimators': 641,
'subsample': 0.8631316254094501}

Results

On comparing the performance metrics of both the models, the XGBoost model produces slightly better results than Random Forest. But the R^2 score of XGB is same as Random Forest for the test dataset. The Mean Absolute error for XGB is 35 minutes, whereas for Random Forest is 36 minutes.

Conclusion

Though the performance of XGB model is satisfactory the model can be improvised further by training the model with the best parameters obtained using the hyper-parameter tuning.
The dataset operations can be accelerated by using cuDF instead of Pandas, which runs on the GPU. Also, the Deep Neural Network models can be used for future study