JOB RUNTIME PREDICTION AND RUNTIME-AWARE SCHEDULING IN HPC SYSTEMS

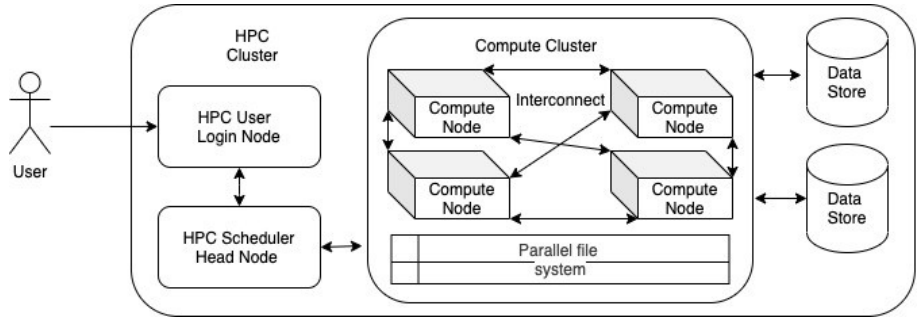
Davide Leone

0 – INTRODUCTION

HPC SYSTEMS

High-Performance Computing (HPC) systems are a class of powerful computers designed to solve complex computational problems that require substantial processing power, memory, and storage resources.

One possible system architecture is HPC Cluster, which consists of multiple nodes connected via a high-speed network. HPC clusters leverage parallel processing, where tasks are divided into smaller sub-tasks that are processed simultaneously across multiple nodes, significantly reducing computation time for large problems.



Simplified version of a typical HPC cluster

The key components of HPC Clusters are:

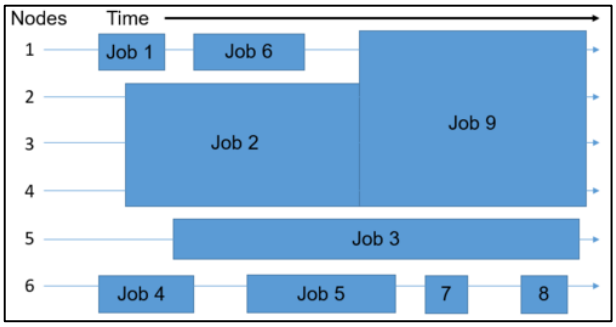
- Head node (aka master node): which is responsible for managing the cluster. It handles job scheduling, resource allocation, and monitoring.
- Compute nodes: which are the primary units performing the actual computations.
- Interconnection: which connects the nodes and enables data exchange (usually high-speed, low-latency networks).
- Storage: which are used to handle the massive data throughput required.

The allocation of resources is crucial as it allows to: reduce waiting times for jobs in the queue, improve performance, balance the workload and reduce energy consumption.

SCHEDULING

Scheduling in HPC clusters is the process of assigning computational tasks (jobs) to the available resources (nodes, CPUs, GPUs, memory) in an efficient and effective manner.

The primary goal of scheduling is to maximize resource utilization, minimize job wait times, and ensure fair access to resources for all users.



In general, scheduling works as follows:

1. Jobs submitted by users are placed in a queue.
2. The scheduler manages the queue, determining the order in which jobs are executed.
3. The order of execution is based on different possible policies and priorities (job size, user priority, estimated job runtime, etc.).
4. The resources are assigned to jobs based on their requirements.

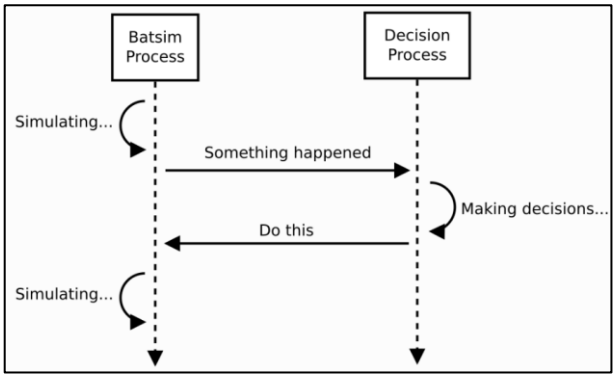
BATSIM

Batsim is a scientific simulator to analyse batch schedulers. A Batsim simulation requires three elements:

1. A platform, which provides functionalities for the simulation of distributed applications.
2. A workload, which are used to define what users want to execute over time (jobs) and how the jobs should be executed (profiles).
3. A scheduler, which determines the order in which jobs are executed.

A Batsim simulation consists in two processes:

- Batsim itself, in charge of simulating what happens on the platform.
- A Decision Process (scheduler), in charge of making decisions.



The two processes communicate via a socket with the synchronous protocol, which follows a simple request-reply pattern. Whenever an event which may require making decision occurs in Batsim in the simulation, the following steps occur:

1. Batsim suspends the simulation.
2. Batsim sends a request to the scheduler (telling it what happened on the platform).
3. Batsim waits for a reply from the scheduler.
4. Batsim receives the reply.
5. Batsim resumes the simulation, applying the decision which have been made.

RESEARCH OBJECTIVES

Usually, jobs submitted to an HPC system are scheduled without prior knowledge on the characteristics of the job. This is probably a limitation, in fact, having the possibility to perform informed scheduling decisions could improve the efficiency of the entire system.

For this purpose, predicting the duration of a job, by analysing only information available when the job is submitted, could be of great importance. Therefore, this work consists of two parts:

1. Job runtime prediction, in which the objective is to test different Machine Learning models on historical data from a real HPC system (Marconi100 from Cineca), to find the best performing model and understand which information is more useful for prediction purposes.
2. Runtime prediction-aware scheduling, in which the objective is to integrate the runtime predictor obtained in the first part into a scheduler (Batsim) for HPC workloads.

DATASET

We used historical data from Marconi100 from Cineca. In particular, we have that:

- The initial dataset has 6’236'346 rows and 100 columns.
- The submission time features are collected (in the column "*tres_req_str*") only for jobs performed between May and November 2020; therefore we consider only the 1’074’576 rows relative to this time period.
- Furthermore, we consider only completed jobs belonging to the "*m100_usr_prod*" partition (which includes more than 99% of the completed jobs), reducing the number of jobs to 887’539.
- Then, we extract the five values stored as strings in the column "*tres_req_str*" into five columns (“*cpu*”, “*mem*”, “*node*”, “*billing*”, “*gres/gpu*”) and we combine them with three other features considered relevant (“*user_id”*, “*qos*”, “*time_limit*”) and the target column ("*run_time*"), obtaining a total of 9 columns on which we will focus our experiments.
- In practice, the columns “*cpu*” and “*billing*” contain the same values, therefore we remove the column “*billing*”, and we remain with 8 columns.
- We also remove the rows where there are NaN, reducing the number of jobs to 631’026.
- Finally, we remove the rows where “*run_time*” is 0 seconds, and we end up with 628’977 jobs, which will be our dataset.

We convert the values in columns to the correct type and standardize the unit of measurement when required. In particular:

- “*mem*” is expressed in MB, GB and TB $\rightarrow$ we convert all to GB.
- “*time_limit*” is expressed in minutes, while “*run_time*” in seconds $\rightarrow$ we express “*run_time*” in minutes.

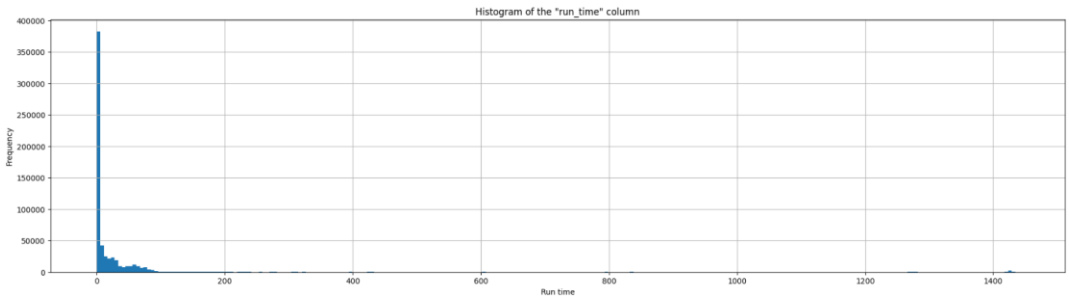
1 – JOB RUNTIME PREDICTION

PRELIMINAR ANALYSIS

	cpu	mem (GB)	node	gres/gpu	user_id	qos	time_limit	run_time
count	628977.000000	628977.000000	628977.000000	628977.000000	628977.000000	628977.000000	628977.000000	628977.000000
mean	121.379298	236.068116	1.693211	5.629788	110.894758	0.051124	1038.068770	43.432751
std	546.656715	1008.593907	6.960532	27.926853	118.594020	0.367796	506.318303	168.718792
min	1.000000	0.097656	1.000000	1.000000	0.000000	0.000000	1.000000	0.016667
25%	4.000000	7.812500	1.000000	1.000000	2.000000	0.000000	720.000000	0.016667
50%	80.000000	230.000000	1.000000	4.000000	93.000000	0.000000	1440.000000	0.083333
75%	128.000000	237.500000	1.000000	4.000000	191.000000	0.000000	1440.000000	22.700000
max	32768.000000	61500.000000	256.000000	1024.000000	387.000000	3.000000	1440.000000	1439.916667

From the table above we can see that:

- Both "*cpu*" and "*mem (GB)*" columns exhibit high variability, as indicated by their large standard deviations and the wide range between minimum and maximum values.
- The data appears to be highly skewed in most columns, with a few extreme outliers significantly affecting the mean values.
- The median values for most columns are lower than the mean, suggesting that a significant number of entries have values much lower than the mean, but a few high values increase the average. This intuition is also clearly visible in the histograms of the columns. As an example, we can see the histogram of the column "*runtime*".



WORKFLOW

We performed a total of four experiments, changing something in the input data at each attempt.

In each experiment we used three models:

- **Decision Tree** regression, which involves splitting the data into subsets based on feature values, creating a tree-like model of decisions. Each internal node represents a decision based on a feature, each branch represents the outcome of a decision, and each leaf node represents a predicted value. The tree is constructed by recursively partitioning the data to minimize a loss function
- **Random Forest** regression, which uses an ensemble of decision trees: each decision tree in the forest is trained on a random subset of the data, with features also randomly selected.

- **Gradient Boosting** regression, which builds a model by sequentially adding predictors (typically decision trees) to minimize a loss function. Each new tree corrects errors made by the previous trees, focusing on the gradients of the loss function with respect to the predictions.

To evaluate our models we used:

- Mean Absolute Error (**MAE**)
- Mean Squared Error (**MSE**)
- Root Mean Squared Error (**RMSE**)
- R^2 score (**R^2**)

To evaluate our models we also checked if the models:

- Overestimate the actual runtime (**OVER-ESTIMATION**). In this case we also calculated the minimum, maximum and average errors and the percentage of cases in which the error is less than one hour.
- Underestimate the actual runtime (**UNDER-ESTIMATION)**. As above, we also calculated the minimum, maximum and average errors and the percentage of cases in which the error is less than one hour. in the following tables)
- Exactly predict the actual runtime (**EXACT-ESTIMATION**), down to half a second.

Finally, we showed how much, as a percentage, the predicted runtime is better than the initial guess ("*time_limit*") at approximating the actual runtime (**EFFECTIVENESS**). We also consider the effectiveness for **Valid Prediction**s, which are the one where the model does not underestimate the actual "*run_time*".

EXPERIMENT 1: BASELINE

DESCRIPTION

We performed regression without any preprocessing of the data, to have a baseline.

EVALUATION

MODEL	MAE	MSE	RMSE	R^2
Decision Tree	23.56	8052.40	89.74	0.72
Random Forest	23.58	7990.01	89.39	0.72
Gradient Boosting	35.89	13146.90	114.66	0.54

MODEL / RESULT	Decision Tree	Random Forest	Gradient Boosting
OVER-ESTIMATION			
Total cases	79.50%	79.64%	80.30%
min(error)	0.01	0.01	0.01
max(error)	1431.00	1351.29	1062.67
avg(error)	14.82	14.78	22.37
error < 60 min	96.30%	96.23%	91.84%
UNDER-ESTIMATION			
Total cases	20.00%	19.89%	19.69%
min(error)	0.01	0.01	0.01
max(error)	1425.53	1425.54	1424.01
avg(error)	58.91	59.35	91.09
error < 60 min	86.67%	86.31%	79.57%
EXACT-ESTIMATION			
Total cases	0.50%	0.47%	0.01%
EFFECTIVENESS			
General	78.10%	78.26%	75.86%
Valid Prediction	97.93%	97.94%	96.13%

COMMENTS

“Decision Tree” and “Random Forest” are both effective, but “Random Forest” has slightly better performance metrics. “Gradient Boosting” is the worst model.

EXPERIMENT 2: BEST USERS PARTITION

DESCRIPTION

We assumed that more experienced users (i.e. those who have submitted more jobs) will be better able to give useful information at submission time. So, we tried to partition the dataset by user to see if considering only a subset of the "best users" gives us better results.

For this purpose, we first calculate the average number of jobs per user and only consider users who have submitted at least 10% of the average number of jobs. We obtain that 126 out of 388 users are “experts”. We then used Decision Tree (the fastest model) to find the 50 best users, which are the one for which the mean squared error between the predicted run time and the actual run time is minimum. We considered the dataset composed by only the best 50 users, which includes almost 80% of all the jobs (497’894 out of 628’977).

EVALUATION

MODEL	MAE	MSE	RMSE	R^2
Decision Tree	8.44	877.96	29.63	0.77
Random Forest	8.42	860.90	29.34	0.78
Gradient Boosting	9.97	964.52	31.06	0.75

MODEL / RESULT	Decision Tree	Random Forest	Gradient Boosting
OVER-ESTIMATION			
Total cases	80.14%	80.19%	81.03%
min(error)	0.01	0.01	0.01
max(error)	1187.33	1185.45	1119.76
avg(error)	5.32	5.29	6.16
error < 60 min	99.88%	99.87%	99.80%
UNDER-ESTIMATION			
Total cases	19.36%	19.32%	18.96%
min(error)	0.01	0.01	0.01
max(error)	1425.49	1425.50	1424.95
avg(error)	21.59	21.64	26.23
error < 60 min	98.23%	98.15%	95.25%
EXACT-ESTIMATION			
Total cases	0.51%	0.50%	0.01%
EFFECTIVENESS			
General	79.38%	79.43%	78.40%
Valid Prediction	98.72%	98.72%	97.23%

COMMENTS

Partitioning the dataset by user and considering only the jobs submitted by the best users improves the results dramatically. It might be interesting to understand what these users did differently, in order to obtain indications on good practices for requesting resources.

EXPERIMENT 3: DATA AUGMENTATION

DESCRIPTION

The previous experiment showed that “expert” users are somewhat better at giving useful information at submission time. So, we tried to enhance the data by adding the average resources requested from each user.

Like in the last experiment, we considered only the user with experience (the one who submitted more than 10% of the average number of jobs submitted per user), obtaining a total of 618’548 jobs.

EVALUATION

MODEL	MAE	MSE	RMSE	R^2
Decision Tree	22.24	7312.82	85.52	0.71
Random Forest	22.26	7275.61	85.30	0.72
Gradient Boosting	26.01	8406.57	91.69	0.67

MODEL / RESULT	Decision Tree	Random Forest	Gradient Boosting
OVER-ESTIMATION			
Total cases	79.90%	79.98%	80.57%
min(error)	0.01	0.01	0.01
max(error)	1425.65	1425.66	1118.25
avg(error)	13.97	13.97	16.20
error < 60 min	96.20%	96.15%	95.64%
UNDER-ESTIMATION			
Total cases	19.69%	19.63%	19.41%
min(error)	0.01	0.01	0.01
max(error)	1425.42	1425.41	1424.76
avg(error)	56.25	56.48	66.72
error < 60 min	87.46%	87.29%	83.63%
EXACT-ESTIMATION			
Total cases	0.41%	0.39%	0.02%
EFFECTIVENESS			
General	78.81%	78.81%	78.81%
Valid Prediction	98.51%	98.51%	98.51%

COMMENTS

Data augmentation slightly improves the results, reinforcing the idea that expert users have a greater ability to request resources.

EXPERIMENT 4: CONSECUTIVE SPLIT

DESCRIPTION

In the last experiment, instead of using a random split to obtain train and test sets we used a consecutive split over time, i.e. all jobs before a certain date are used for training and all those after are used for testing.

We chose the date such that the test set contained exactly the same number of jobs as the one with random split.

We considered as input the "base" dataset of experiment two, therefore the one without any consideration of the "expert" users.

EVALUATION

MODEL	MAE	MSE	RMSE	R^2
Decision Tree	8.33	3438.32	58.64	0.62
Random Forest	8.35	3432.47	58.59	0.62
Gradient Boosting	22.90	5086.70	71.32	0.44

MODEL / RESULT	Decision Tree	Random Forest	Gradient Boosting
OVER-ESTIMATION			
Total cases	94.40%	94.86%	95.99%
min(error)	0.01	0.01	0.02
max(error)	1196.12	1184.39	722.10
avg(error)	4.18	4.08	16.96
error < 60 min	99.44%	99.13%	98.90%
UNDER-ESTIMATION			
Total cases	5.25%	5.13%	4.00%
min(error)	0.01	0.01	0.02
max(error)	1425.71	1425.71	1399.33
avg(error)	83.43	87.44	165.44
error < 60 min	81.69%	80.87%	67.75%
EXACT-ESTIMATION			
Total cases	0.35%	0.01%	0.00%
EFFECTIVENESS			
General	94.22%	94.27%	93.40%
Valid Prediction	99.45%	99.37%	97.79%

COMMENTS

Unexpectedly, the results seem better than those obtained with random split. We can see that all the error values are much better, while those of R2 are worse, indicating that the quality of the models has decreased despite an average better predictive capacity.

These results can be explained by the fact that, using the consecutive split, in the test set there are much lower average runtime values than in the training set. Furthermore, the standard deviation of all the test set columns is smaller than that of the training set columns.

CONCLUSIONS

In all cases, it seems that the values predicted by the models are better at approximating the "*run_time*" than "*time_limit*". In particular:

- When the models overestimate the runtime (on average around 80% of total cases), this results in almost a 98% improvement (on average).

- When the models underestimate the runtime (on average around 20% of total cases), the job would be interrupted before being completed. This is obviously unacceptable, given that in the original dataset, the percentage of jobs interrupted for reaching "*time_limit*" is 1.40%.
    - In about 85% of these “underestimation” cases, a simple solution could be to add 60 minutes to the predicted runtime, reducing the number of jobs that would be interrupted before finishing to less than 3% of the total (which is still a lot but more in line with the original value of 1.40%). Using this “safe” prediction, we have that the models overestimate the runtime 97% of the time (obviously with higher average error), but we still have a more than 91% improvement (on average) with respect to “*time_limit”*.

2 – RUNTIME AWARE SCHEDULING

DATA PREPARATION

We start from the dataset used in the first part of the project, containing 628'977 jobs. The only difference is that for this part, since Batsim uses seconds as unity of time, we expressed "*time_limit*" in seconds and kept "*run_time*" as it was. During our experiments we noticed that the unity of measurement does not impact the capacity of our models to predict the runtime. Then, we divided the dataset in two parts:

- `df_train`, which contains most of the data and is used to train the regressor used to predict the runtime. Then, we saved it into a .csv files called "*train_jobs.csv*".
- `df_sched`, which contains the data relative to the last 24 hours stored in the original dataset (a total of 4'407 jobs). `df_sched` also includes `sumission_time`, which we manipulated to ensure that the first job in chronological order is submitted at time 0 seconds. We also changed the jobs that require a high number of CPUs, capping them at 256, to ensure that the platform used for the simulation is able to handle the request. Then, we saved it into a .csv files called "*sched_jobs.csv*".

SIMULATION INPUTS

For the platform we used:

- One of the defaults BatSim platforms, defined in the file *"cluster512.xml"*, which has a total of 512 resources available.
- A modified version of *"cluster512.xml"*, called *"clusterMarconi.xml"*, which has a total of 15'680 resources available (which should allow to consider multiple jobs in parallel).

For the workload we defined a function in python, that takes a file .csv and create a file .json which is structured as required for a workload. The logic of this function is very naïve:

- The jobs are directly created using the information given by the input file, as follows:

```
job = {
    "id": job_id,
    "subtime": float(row['submission_time']),
    "res": int(row['cpu']),
    "profile": profile_name,
    'walltime' = int(row['time_limit']),
    "user_id": int(row['user_id']),
    "qos": int(row['qos']),
    "node": int(row['node']),
    "gpu": int(row['gres/gpu']),
    "cpu": int(row['cpu']),
    "mem": float(row['mem (GB)']),
    "time_limit": int(row['time_limit'])
}
```

  Where `job_id` is simply an integer which increases every time a new row is considered, and `profile_name` is a string which represents a profile.

- The profiles are all of the simplest profile type, called "delay", which is only a fixed number of seconds during which the machines will sleep. We defined a profile for each combination of cpu and runtime

(a total of 1807 profiles), we called it "`D_{cpu}_{runtime}`" and we assigned it to the corresponding job so that its runtime is the same as the runtime recorded in our dataset.

We created a workload, relative to "*sched_jobs.csv*", called "*slurm_workload_24h.json*".

SCHEDULING ALGORITHM

To implement the scheduler we started from one defined in `pybatsim` by the class `FillerSched`. We modified this scheduler to prioritize jobs with smaller predicted runtimes, in the following way:

1. Train a runtime prediction model, with a method called `train_regressor`, which loads historical job data from the file "*train_jobs.csv*", trains a Decision Tree Regressor, and stores in `self.regressor`. This is done only once at the beginning of the simulation (`onSimulationBegins`).
2. Predict the runtime of each job upon submission, with the `onJobSubmission` method, where the runtime prediction is made, and `job.reqtime` is set to the predicted value.
3. Prioritize jobs based on their predicted runtimes when scheduling, with the `scheduleJobs` method, where jobs in `openJobs` are sorted based on `job.reqtime` to give priority to jobs with smaller predicted runtimes.

We defined our scheduler in the file *"runtimePredictionAware.py"* as follows and called our class `RuntimePredictionAwareScheduler.`

runtimePredictionAware.py

```
class RuntimePredictionAwareScheduler(BatsimScheduler):
    # Called when the simulation starts: initializes some variables and trains the regressor using historical job data
    def onSimulationBegins(self):
        self.nb_completed_jobs = 0

        self.jobs_completed = []
        self.jobs_waiting = []

        self.sched_delay = 0.005

        self.openJobs = set()
        self.availableResources = ProcSet((0, self.bs.nb_compute_resources - 1))

        self.jobs = []
        self.regressor = None
        self.train_regressor()

    # Loads historical job data from a CSV file and use them to train a Decision Tree Regressor model
    def train_regressor(self):
        try:
            logging.info("Loading historical data..")
            df_train = pd.read_csv("train_jobs.csv")
            logging.info("Loading complete!")

            X_train = df_train.drop(columns=['run_time'])
            y_train = df_train['run_time']

            self.regressor = DecisionTreeRegressor(random_state=13)
            logging.info("Training the regressor..")
            self.regressor.fit(X_train, y_train)
            logging.info("Training complete!")
        except Exception as e:
            logging.error(f"Error in training regressor: {e}")
```

```
# Predicts the runtime of a job using the trained regressor
def predict_runtime(self, job):
    try:
        job_dict = job.json_dict

        # Extract submission features from the job
        features = {
            'cpu': job_dict['cpu'],
            'mem (GB)': job_dict['mem'],
            'node': job_dict['node'],
            'gres/gpu': job_dict['gpu'],
            'user_id': job_dict['user_id'],
            'qos': job_dict['qos'],
            'time_limit': job_dict['time_limit']
        }

        # Convert the features into a DataFrame
        df = pd.DataFrame([features])
        # Predict the runtime using the trained model
        predicted_runtime = self.regressor.predict(df)

        return predicted_runtime[0]
    except Exception as e:
        logging.error(f"Error in predicting runtime: {e}")
        return float('inf')  # Return a large value to deprioritize in case of error

# Schedules jobs based on their predicted runtimes
def scheduleJobs(self):
    scheduledJobs = []

    print('openJobs = ', self.openJobs)
    print('available = ', self.availableResources)

    # Sort openJobs based on predicted runtimes
    sorted_openJobs = sorted(self.openJobs, key=lambda job: job.reqtime)

    # Iterating over a copy to be able to remove jobs from openJobs at traversal
    for job in sorted_openJobs:
        nb_res_req = job.requested_resources
        # Allocates resources to jobs if available
        if nb_res_req <= len(self.availableResources):
            # Retrieve the *nb_res_req* first available resources
            job_alloc = ProcSet(*islice(self.availableResources, nb_res_req))
            job.allocation = job_alloc
            scheduledJobs.append(job)
            # Updates the available resources
            self.availableResources -= job_alloc

            self.openJobs.remove(job)

    # update time
    self.bs.consume_time(self.sched_delay)

    # Executes the scheduled jobs
    if len(scheduledJobs) > 0:
        self.bs.execute_jobs(scheduledJobs)

    print('openJobs = ', self.openJobs)
    print('available = ', self.availableResources)
    print('')

# Called when a new job is submitted: predicts the job's runtime and adds it to the set of open jobs
def onJobSubmission(self, job):
    # Reject the job if it requests more resources than the machine has
    if job.requested_resources > self.bs.nb_compute_resources:
        self.bs.reject_jobs([job])
    else:
        # Predict runtime and set it for the job
        predicted_runtime = self.predict_runtime(job)
        # Set the predicted runtime as the required time
        job.reqtime = predicted_runtime
        # Adds the job to the set of open jobs
        self.openJobs.add(job)
```

```
# Called when a job is completed: releases the resources allocated to the job
def onJobCompletion(self, job):
    self.availableResources |= job.allocation

# Called when there are no more events to process: ensure any remaining jobs are scheduled
def onNoMoreEvents(self):
    self.scheduleJobs()
```

SETUP AND START

To use our scheduler (`pred_aware`), we have to register it in the `pybatsim` entry point by adding the following line in the *"pyproject.toml"* file, under the `[tool.poetry.plugins."pybatsim.schedulers"]` section.

```
pred_aware = "runtimePredictionAware:RuntimePredictionAwareScheduler"
```

which follows the pattern:

```
<scheduler_name> = <file_name>:<Scheduler_class_name>
```

Once done, to use the scheduler we have to create a file *"my_shell.nix"* (modifying the file *"default.nix"* defined in the root folder of Pybatsim) to define a virtual environment with everything needed to launch the scheduler. In particular we added lines to allow the use of `pandas` and `sklearn`.

At this point we are ready to enter the virtual environment by running:

```
nix-shell ../my_shell.nix -A my-shell
```

And then launch the scheduler with:

```
pybatsim pred_aware
```

WORKFLOW

We performed a total of four experiments, aggregated in couple:

- Experiments 1 and 2
  - platform = *"cluster512.xml"*,
  - workload = "*slurm_workload_24h.json",*
  - schedulers = {`pred_aware`, `easy_bf`}
- Experiments 3 and 4
  - platform = *"clusterMarconi.xml"*,
  - workload = "*slurm_workload_24h.json",*
  - schedulers = {`pred_aware`, `easy_bf`}

The results of our experiments are stored in a directory called *"/tmp/expe-out"*, that was created beforehand. The output files are:

- *"out_jobs.csv"*, which contains information about the execution of each job.
- *"out_schedule.csv"*, which contains aggregated information about the whole simulation (such as makespan, mean waiting time, etc.).
- *"out_machine_states.csv"*, which is a time series about the platform usage that stores how many machines are in each state for each time interval.

First, we compared the aggregated information about the whole simulation (*"out_schedule.csv"*) for each couple of experiments.

Then, we compared the percentage of jobs that wait less than some arbitrarily chosen time intervals (one minute, ten minutes, one hour, six hours and one day).

Finally, we merged the results to compare waiting time for corresponding jobs.

OUTPUT DETAILS

In the following, we have that:

- **makespan** is the completion time of the last job.
- **scheduling_time** is the time (in seconds) spent in the scheduler.
- **mean_waiting_time** is the average waiting time observed on jobs. The waiting time of a job is:

  $$waiting\_time = starting\_time - submission\_time$$

  It is the amount of time a job spends waiting in the queue before it starts executing. Lower waiting time is what we want to achieve.
- **mean_turnaround_time** is the average turnaround time observed on jobs. The turnaround of a job is:

  $$turnaround\_time = finish\_time - submission\_time$$

  It includes both the time the job spends waiting in the queue and the time it spends executing. It reflects the efficiency of the system in handling jobs.
- **mean_slowdown** is the average slowdown observed on jobs. The slowdown of a job is:

  $$slowdown = \frac{turnaround\_time}{execution\_time}$$

  It is a measure of how much longer a job takes to complete compared to its actual execution time. It is useful for understanding how scheduling affects the performance of individual jobs.
- **max_waiting_time** is the maximum waiting time observed on a job.
- **max_turnaround_time** is the maximum turnaround time observed on a job.
- **max_slowdown** is the maximum slowdown observed on a job.

EXPERIMENTS 1-2: BASELINE

The first couple of experiments were performed on a platform with a total of 512 resources (*"cluster512.xml"*), and on a workload containing 4'407 jobs referring to a 24-hour period ("*slurm_workload_24h.json"*).

To perform the experiments we simply opened two terminals and launched the following commands:

TERMINAL 1

```
batsim -p /tmp/batsim-src-stable/platforms/cluster512.xml -w /tmp/batsim-src-stable/workloads/slurm_workload_24h.json -e "/tmp/expe-out/out"
```

TERMINAL 2

```
nix-shell ../my_shell.nix -A my-shell
pybatsim pred_aware
```

```
batsched -v easy_bf
```

EVALUATION

	PRED_AWARE	EASY_BF	IMPROVEMENT
makespan	1029198.211600	1090869.293818	-5.99%
scheduling_time	210.345998	194.504246	+7.53%
mean_waiting_time	127474.557009	163846.310703	-28.54%
mean_turnaround_time	128979.200758	165350.954452	-28.21%
mean_slowdown	22785.239851	20097.171103	+11.80%
max_waiting_time	994211.211600	1026349.259818	-3.21%
max_turnaround_time	1024094.211600	1027933.289418	-0.37%
max_slowdown	450511.649074	1026350.260057	-128.09%

SCHEDULER / WAITING TIME	PRED_AWARE	EASY_BF
< 1 minutes	13.09%	2.06%
< 10 minutes	22.76%	2.95%
< 1 hour	32.92%	15.95%
< 6 hours	55.84%	56.93%
< 24 hours	66.94%	71.84%
> 24 hours	33.06%	28.16%

COMMENTS

We can observe that, using the `pred_aware` scheduler, there are some clear improvements over the `easy_bf` scheduler:

- The total time required to go through all jobs in the workload is 6% lower.
- The mean waiting time of a job is more than 28% lower.
- The waiting time is very low (less than 10 minutes) for almost 8 times more jobs (1004 vs 130).

On the other hand, using the `pred_aware` scheduler, the waiting time is very high (more than 1 day) for almost 5% more jobs than when using the `easy_bf` scheduler (1457 vs 1241).

In general, we can notice that the number of available resources is a big bottleneck. This is supported by the fact there are 689 jobs that required the maximum of 256 resources and 685 of them are executed in the last 700 jobs. This increases the average and maximum waiting time, because the jobs that require too many resources are put on hold until all the other jobs are completed and then are executed in series. This was the main reason we had to increase the number resources available in the second pair of experiments.

EXPERIMENTS 3-4: BIG PLATFORM

The second couple of experiments were performed on a platform with a total of 15'680 resources (*"clusterMarconi.xml"*), and on the same workload of the experiments 3-4 (*"slurm_workload_24h.json"*).

Again, to perform the experiments we simply opened two terminals and launched the following commands:

TERMINAL 1

```
batsim -p /tmp/batsim-src-stable/platforms/clusterMarconi.xml -w /tmp/batsim-src-stable/workloads/slurm_workload_24h.json -e "/tmp/expe-out/out"
```

TERMINAL 2

```
nix-shell ../my_shell.nix -A my-shell
pybatsim pred_aware
```

```
batsched -v easy_bf
```

EVALUATION

	PRED_AWARE	EASY_BF	IMPROVEMENT
makespan	86272.006800	86272.002400	+0.00%
scheduling_time	37.844925	240.739561	-84.28%
mean_waiting_time	846.539090	953.381263	-11.21%
mean_turnaround_time	2351.182838	2458.025012	-4.35%
mean_slowdown	2.308898	45.851883	-94.96%
max_waiting_time	17003.092800	12608.038400	+34.88%
max_turnaround_time	64657.006800	64657.002400	+0.00%
max_slowdown	261.081800	12156.040600	-97.85%

SCHEDULER / WAITING TIME	PRED_AWARE	EASY_BF
< 1 minutes	84.86%	80.58%
< 10 minutes	88.56%	84.41%
< 1 hour	91.49%	89.88%
< 6 hours	100.00%	100.00%

Furthermore, with respect to the `easy_bf` scheduler, using the `pred_aware` scheduler:

- The waiting time is at most the same (+0.01 seconds) 3992 out of 4407 times (90.58%).
- The waiting time is reduced by at least 10 minutes 363 out of 4407 times (8.24%).
- The waiting time is reduced by at least 1 hour 199 out of 4407 times (4.52%).
- The waiting time is increased by at least 10 minutes 253 out of 4407 times (5.74%).
- The waiting time is increased by at least 1 hour 153 out of 4407 times (3.47%).

COMMENTS

We can observe that, using the `pred_aware` scheduler, there are some clear improvements over the `easy_bf` scheduler:

- The mean waiting time of a job is more than 11% lower.
- The waiting time is very low (less than 10 minutes) for more than 4% more jobs (3903 vs 3720).
- The waiting time is at most the same in more than 90% of the cases (3992 out of 4407) and is significantly lower (1 hour) in more than 4.5% of the cases (199 out of 4407).

On the other hand, using the `pred_aware` scheduler, the maximum waiting time is higher than that obtained using the `easy_bf` scheduler by a significant margin (almost 35%). But the percentage of jobs in which the waiting time is higher than the maximum waiting time of `easy_bf` is less than 2% of jobs (85 out of 4407).

Furthermore, the total time required to go through all jobs in the workload is the same, because in this case it is limited only by the execution time of all the jobs.

CONCLUSIONS

In general, we can say that using the `pred_aware` scheduler the waiting time is reduced with respect to the `easy_bf` scheduler. However there is also a non-negligible number of jobs for which the waiting time is increased.

We can conclude that this work indicates the possibility of improving resource management in an HPC cluster, with the aim of reducing the average waiting time.

Possible future extensions to this work include:

- Improvements to the `pred_aware` scheduler (better regression model to predict runtime, more complex logic to assign priority to jobs, etc.).
- Comparisons with other predefined schedulers besides `easy_bf`.
- Application of the scheduler to data coming from other HPC systems, possibly more modern than Marconi100.