Estimating an HPC Facility's Capacity For Accommodating Real-time Workflows

LAWRENCE BERKELEY NATIONAL LABORATORY

ADVANCED TECHNOLOGY GROUP - NATIONAL ENERGY RESEARCH SCIENTIFIC COMPUTING (NERSC) CENTER

(06/03/2019 - 08/19/2019)

Submitted By

Eashan Adhikarla

Research Intern - 062124

Supervisor:

Nicholas James Wright

Mentor:

Brian Austin

ACKNOWLEDGMENT

The incredible opportunity I had to intern with **Dr. Brian Austin** has been a great learning experience, paving the way for both professional development and personality enrichment. Therefore, I consider myself immensely privileged having been a part of it. I extend my sincere gratitude to all the Staff members and Postdocs who led me through this internship.

I would like to profoundly thank **Dr. Brian Austin** for recognizing my skills and abilities and helping me to put them into diligent practice. Working with him has been immense and I am glad to assimilate a wealth of takeaways from him and this lab. He has undeniably been the greatest mentor in my career so far. Brian has not only educated me about the expertise pertinent in the field of Engineering and Research but also the application of these principles in the real world where I am striving to constantly escalate the values of life.

I would like to express my deepest gratitude and mention a special acknowledgment to **Dr. Nicholas Wright** who, despite being exceptionally occupied with his own liabilities, took time out to hear, check, and guide me down the correct path, allowing me to propose and exercise my ideas for the project - Workflow Analysis.

I see this window of opportunity as a major milestone in the growth of my career. I promise to endeavor to make the best possible use of my acquired abilities and expertise, continuing nevertheless to work on their enhancement for achieving the required career goals. I would be greatly obliged to work with the ATG group in the future again.

I. ABSTRACT

With the exponential growth of supercomputers in parallelism, applications are growing more diverse. This includes traditional large-scale HPC MPI jobs and ensemble workloads such as finer-grained many-task computing (MTC) applications. The growing number of workloads, including tightly-integrated experimental facilities and emergency decision-making tasks, require response-times that are not currently compatible with typical queue wait-time policies. Hence, any policy to accommodate real-time requirements will impact system utilization and disrupt other users. The project estimates the consequences of three potential real-time scheduling policies to analyse and estimate the HPC facility's impact on an incoming real-time job on NERSC's Cori Supercomputer. These proposed policies tend to improve the allocation procedure by immediately allocating real-time jobs without interrupting the long run batch jobs in the memory.

II. INTRODUCTION

Supercomputers are extremely contended mainframe computers used for scientific calculations. All work on the supercomputer is presented as a batch of tasks. These batch activities are extremely complex as science computations are extremely big calculations with petabytes of information. The computational resources for these applications on our supercomputer are accessed through a scheduler. The scheduler we use is Simple Linux Utility for Resource Management (SLURM) and it is an open-source scheduler for Linux and Unix-like kernels that schedules and tracks batch tasks. These tasks are submitted as jobs and just about all programs are run as batch tasks on the Cori supercomputer.

All the batch tasks in Cori are managed by the batch System. Managing it is a challenging process as the tasks include complicated SPMD instructions i.e. Single Program Multiple Data. Each instruction coming in as a batch task is divided into multiple tasks and given to different processors to solve the overall task in parallel. In all, there are more than 50,000 SPMD instructions running per day on Cori. Hence the scheduling process for batch tasks' is highly concurrent in nature and overloads itself with a lot of other emergency issues. The HPC resources are limited and all the instructions/tasks are highly demanding which is the biggest load factor to be reduced as much as possible.

Our research solely focuses on real-time jobs. Various kinds of jobs are entering the Cori supercomputer every day. Two main kinds of jobs for our research purpose are Batch Jobs (which is also referred to as a normal job in the writing) and real-time jobs. Real-time jobs are jobs that are urgent and need to be responded to immediately. Apart from real-time jobs, all other kinds of jobs in our case are considered normal batch jobs. Examples of real-time jobs are jobs Advanced Light Source (ALS) submitting jobs with petabytes of beamline data from the cyclotron and Satellite data for measuring angles from the Computational Resource division.

Upgrading and changing the policies in an application to architecture is a challenge, not only in porting the code but also in understanding and tuning the performance. Rather than manually performing the analysis, people tend to use tools to motivate optimization. Therefore approaches to simulate the real-time environment and observing the performances are becoming one of the most critical components in the modern policy designs. These proposed policies have been impactful on hypothetical real-time jobs used in the simulation process, increasing the performance of the system and cutting down the waste by canceling the jobs and reducing the idle node hours per node request.

A lot of research shows how we can add time-sharing HPC workload and space-sharing workload to reduce the total time from submission to till the completion of the job. This research is focused on understanding and estimating the impact of the new policies designed with time and memory as a constraint on system utilization. Also, how can we immediately allocate resources to incoming real-time jobs without making them wait and not letting other batch jobs suffer because of real-time jobs? The current policies either immediately allocate the resources to the real-time job by killing a fraction of normal jobs running in the memory or it makes the real-time job wait in the draining process. Our analysis shows how to combat this situation, and also the answers to these upgrades are minimal effort policy upgrades.

III. PROGRESS & RESULTS

In this section, we will cover new strategies to overcome the problem discussed above. Firstly, why and how the data was chosen for this research and each of the strategies in detail with observations.

DATA COLLECTION

The importance of data collection was to maintain a realistic scenario in the simulation process. Instead of making our own randomized data we collected the data from the actual users. The data we fetched from the SLURM database belongs to the old and current NERSC users between the dates 11/01/2017 and 11/02/2018.

We have analyzed three distinct real-time queue strategies in this research work for incoming real-time jobs. This is performed on the Cori Supercomputer for KNL architecture and Haswell architecture. Overall, analyses show the impact and scale of improvement by answering some meaningful questions like, How much time does a real-time job have to wait after submitting the job into the Cori supercomputer? How much time does the nodes waste being Idle nodes while in the draining process? How much time does the job preemption policy waste by canceling the running batch jobs from the memory on arrival of the real-time job? How much memory pressure is created on the job pausing policy when the real-time job comes into the memory?

METHODOLOGY

We obtained from NERSC, a performance log with details including the number of nodes used for the job, start time of the job, end time of the job, memory footprint, maximum resident set size (size of memory the job reserved in the runtime), the architecture of the job (KNL, Haswell, Shared) for all jobs that ran on Cori's nodes during 2018. These performance logs are generated daily in the and contain petabytes (1000s of terabytes) of data. These logs are a part of the DB01 database which is a subset of the Slurms' original database and is used for research purposes. We randomly sampled 50 time points that correspond to the submission times for a set of hypothetical real-time jobs. Job logs were then analyzed to compute the median wait-time, utilization-loss and memory availability for each of the proposed scheduling policies described below. Increasing the time-points for the analysis is always useful for accuracy but, because of time constraints, 50 time-points were used for each simulation.

The table below helps in understanding the present workload on the Cori supercomputer on its compute nodes. The workload ranges from 1000 jobs to 60,000 jobs every day, with MPI running going over 50,000 a few times, which increases the overload on Cori. The total number of compute nodes 12076 and is limited to any number of jobs coming in. The brief outlook for this configuration is shown below[10];

Cori and its Workload at Glance	
Compute Nodes	9688 (KNL), 2388 (Haswell)
Total Cores	658,784 (KNL), 76,416 (Haswell)
Jobs per day	60,000
Nodes per Job	1-9688, Avg.= 716 hours
Time per Job	<1min - 8 days, Avg.= 19 hours

Table 1. Cori and its Workload at Glance

POLICIES

In order to perform the simulation experiment we came up with three novel policies (Draining, Job Cancelation, Memory Availability), tested them in a simulated environment and discussed the results and impact of system utilization.

POLICY 1 - DRAINING:

The first policy we considered is based on measuring the impact on the waiting time of the supercomputer during an incoming real-time job as a result of which the system draining process begins. The policy states— *Upon submission of a real-time job, the compute nodes are drained by allowing currently running batch jobs to complete normally and prohibiting batch jobs from starting until the real-time job begins.*

For instance in the figure below, If point ‘**a**’ is the submission time of the real-time job, then no batch jobs will start until the real-time job starts at point ‘**b**’.

Draining has two undesirable consequences.

1. The real-time job does not start immediately.
2. Some nodes are left unused (Idle) while waiting until there are enough free nodes available for the real-time job to start.

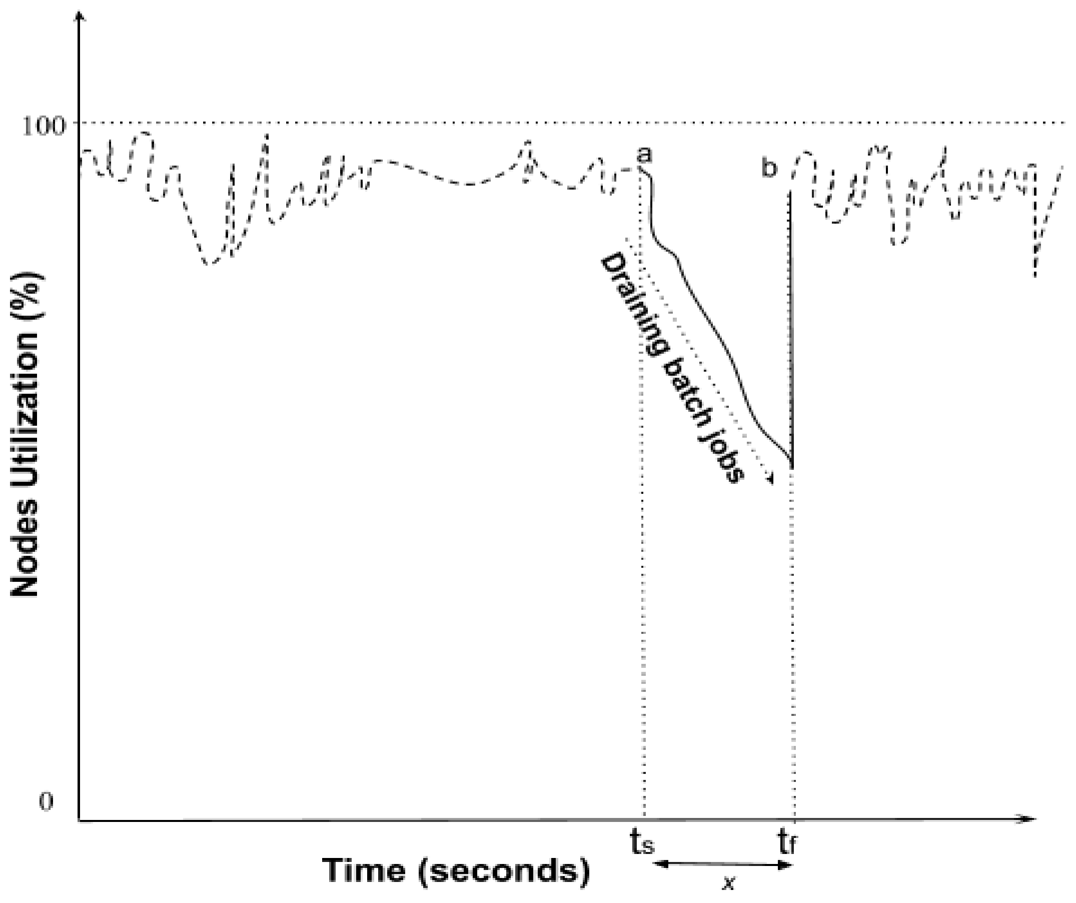

Fig.1. Showing the draining process of batch jobs from point a to point b to release busy nodes for real-time jobs.

Equation (1) below describes that at a particular time stamp 't', t_s being the arrival time of the real-time job j, available idle nodes for allocating it to the real-time job. So, with this equation we can measure if the the available nodes are less than the required nodes for the real-time job then how long the real-time job has to wait in the draining process before getting started. This impact will help us in understanding wait time metrics well for any real-time job.

Nodes available at time t is the vacancy of nodes among the total number of nodes in each architecture (KNL, Haswell). Here, we are convoluting the step function with jobs running at time t and to consider only those jobs from left to right in the queue Fig. 1. until we reach the requirement of the real-time job in the draining process. Adding n_0 to the final value as there are a few unused nodes in the system.

$$N(t;t_s) = n_0 + \sum_{j}^{\prime} n_j * H(t - t_{f_j})$$

$$\textit{Nodes available}(t) = n_0(t_s) + \sum\nolimits'_{j=0} n_j * \begin{cases} 0, & t_{f_j} > t \\ 1, & t_{f_j} \leq 0 \end{cases} \quad \textbf{Eq (1)}$$

Let's look at the wait time metrics described in the equation (1). Once you have the wait time for every instance (for every value of the node request as in Fig. 2.a.), we can also calculate how many nodes were idle at the time of draining process.

In the Equation (2) below, Utilization loss at time *t* is total idle-hours while draining the system Fig. 1. when the nodes were neither being utilised by the batch jobs nor by the real-time jobs, they were eventually wasted. Fig. 2.b. is shown by idle node hours per node requested. We calculate the area of idle nodes per node requested by convoluting the time difference in the x-axis (i.e., arrival time of real-time job and finish time of batch job) with current running jobs. This convolution is done only with the current running jobs in the stack from top to bottom until we reach the required idle node value. We have added n_0 with idle-node area to the final value as realistically there are approximately 2% nodes which are not utilized at all.

$$N(t; t_s) = n_0 * (t - t_{f_j}) + \sum_{j}^{\prime} n_j * (t - t_{f_j}) * H(t - t_{f_j})$$

$$\textit{Utilization loss by Idle Nodes } (t) = n_0 * (t - t_s) + \sum\nolimits'_{j=0} n_j * (t - t_{f_j}) * \begin{cases} 0, & t_{f_j} > t \\ 1, & t_{f_j} \leq 0 \end{cases} \quad \textbf{Eq (2)}$$

Here,

n_0	=	Idle nodes at the time of real-time job submission
t_f	=	Completion time for normal jobs.
j	=	Index of “normal” jobs running at t_0.
t	=	Submission of real-time job in the queue.

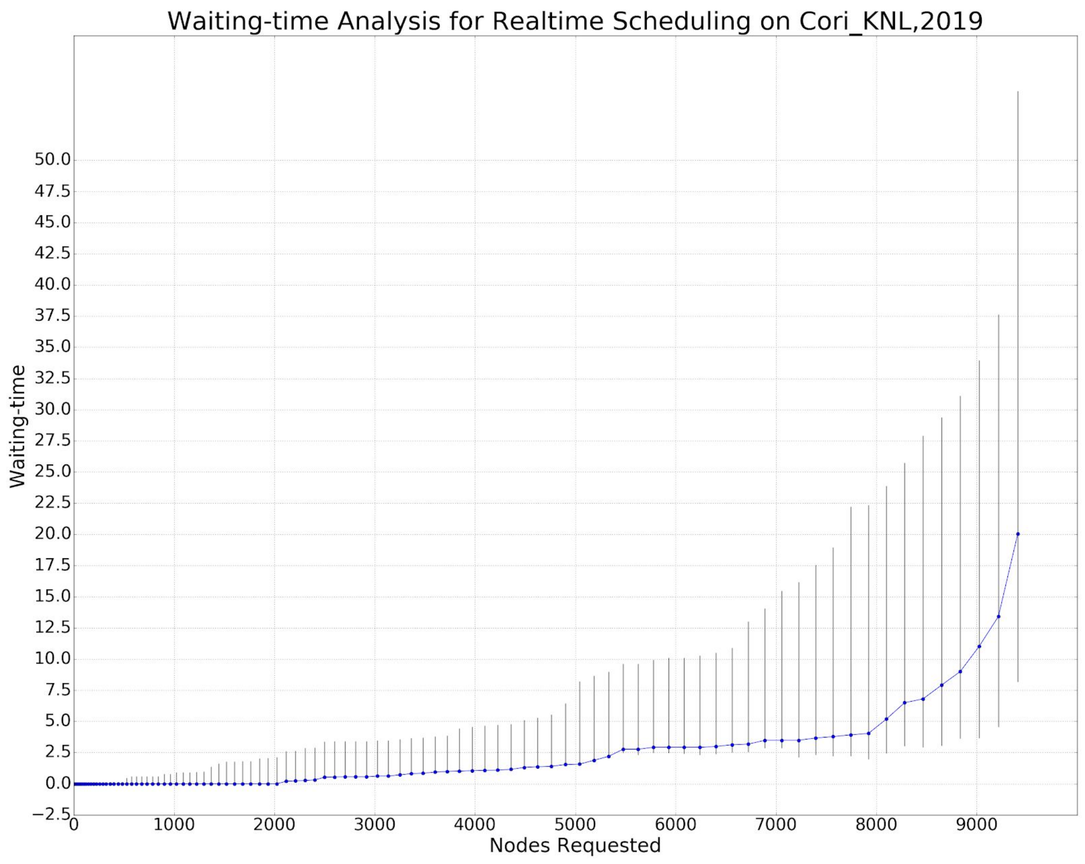

Fig. 2.a. Impact on waiting-time (in hours) for the real-time job during the system draining procedure to acquire N-nodes at time t.

For the Fig. 2.a. Until 2048 nodes there is almost no wait-time if a real-time job just came in and demanded nodes. A useful metric to note here would be the error bars in the plot. The error bars covers the 25th percentile (lower limit) and the 75th percentile (Upper limit) of the median 50th percentile value. This range from the lower limit to the upper limit is the inner-quartile range (IQR) for the waiting time in hours. This plot shows the centred position and the spread across the error bars which overall summarizes the result at each point in x-axis very well.

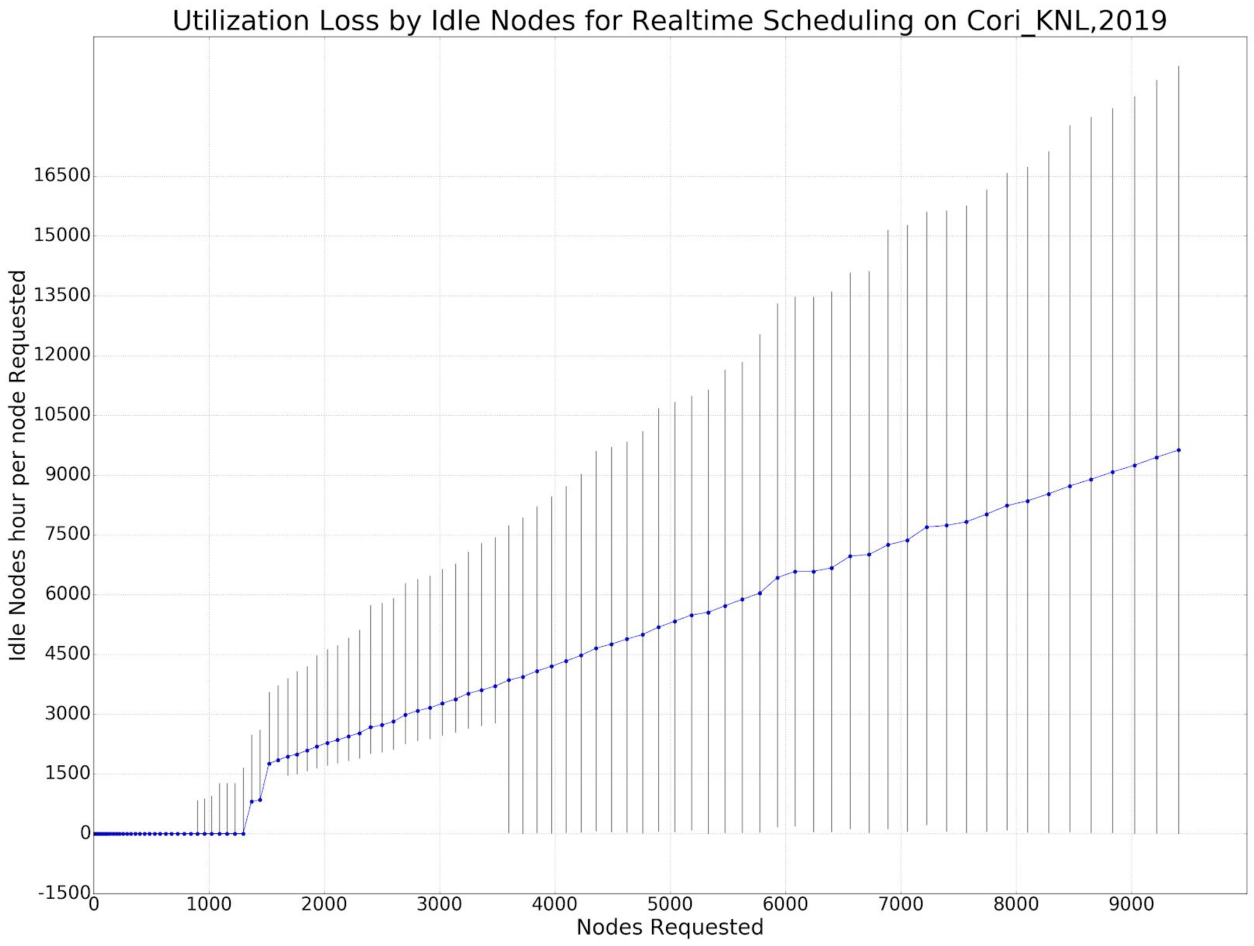

Fig. 2.b. Node-hours wasted by the Idle Nodes while draining the system for every arrival of a real-time job requiring N-nodes at time t.

For the Fig. 2.b. It looks like at node request value around 1024 — Idle node hours increases frequently which means the system is draining a lot at that point for the real-time job. So for a node request value of less than 1024 there is very little impact on system utilization when an incoming job demands for the compute nodes that Cori can easily provide.

POLICY 2 - JOB CANCELLATION:

The second policy is to consider measuring the impact on the system's time and computational waste when we cancel a fraction of running "normal" jobs on the arrival of real-time jobs. This policy states— *Upon submission of a real-time job, a fraction of the currently running batch jobs are canceled to allow the real-time job to start immediately. Jobs are selected for the cancelation to minimize waste of computational resources. Also, computational resources are* *limited to space and can be made only by canceling jobs. This policy is extremely disruptive to users whose jobs are canceled.*

From a normal NERSC user's perspective, the policy is extremely irritating as the normal jobs are terminated as soon as the real-time job arrives. But this policy definitely does not waste a lot of hours, days and computational energy by preempting the job. Also, jobs with low priority queues can be requeued back and while jobs in a medium priority queue may be suspended. Despite being disruptive, it is less disruptive to the current approach used by NERSC.

The useful observation to note here is that, on each node request value, we know the amount of time we waste in hours. So we can now constraint to cancel only those jobs in the queue which immediately started or maybe a few minutes back (this value can be set in the policy). So that we prevent those "normal" jobs being canceled/preempted which had been running for hours and days. In this way, we do not waste a lot of time and computational power used by the Cori supercomputer.

Equation (3) below describes, the Hours wasted at time t which is the total hours wasted on an arrival of real-time job. Here, the time difference (t - t_{sj}) is the difference between run time of the normal batch job and arrival of the real-time job which is basically how long did the batch job run before getting canceled. This run-time is convoluted with respective job j which is canceled/preempted from the memory. The summation of all such jobs gives the total hours wasted by each architecture.

$$Wasted = \sum_{j}^{\prime} n_j * (t_0 - t_{s_j})$$

$$Hours\ wasted\ (t) = \sum\nolimits_{j=0}^{\prime} n_j * \left(t - t_{i_j}\right) \quad \textbf{Eq (3)}$$

Where,

j	=	Index of batch jobs running at time 't'.
t_{ij}	=	Time when the normal batch-job 'j' started
n	=	Number of Iterations
t	=	Submission of real-time job in the queue

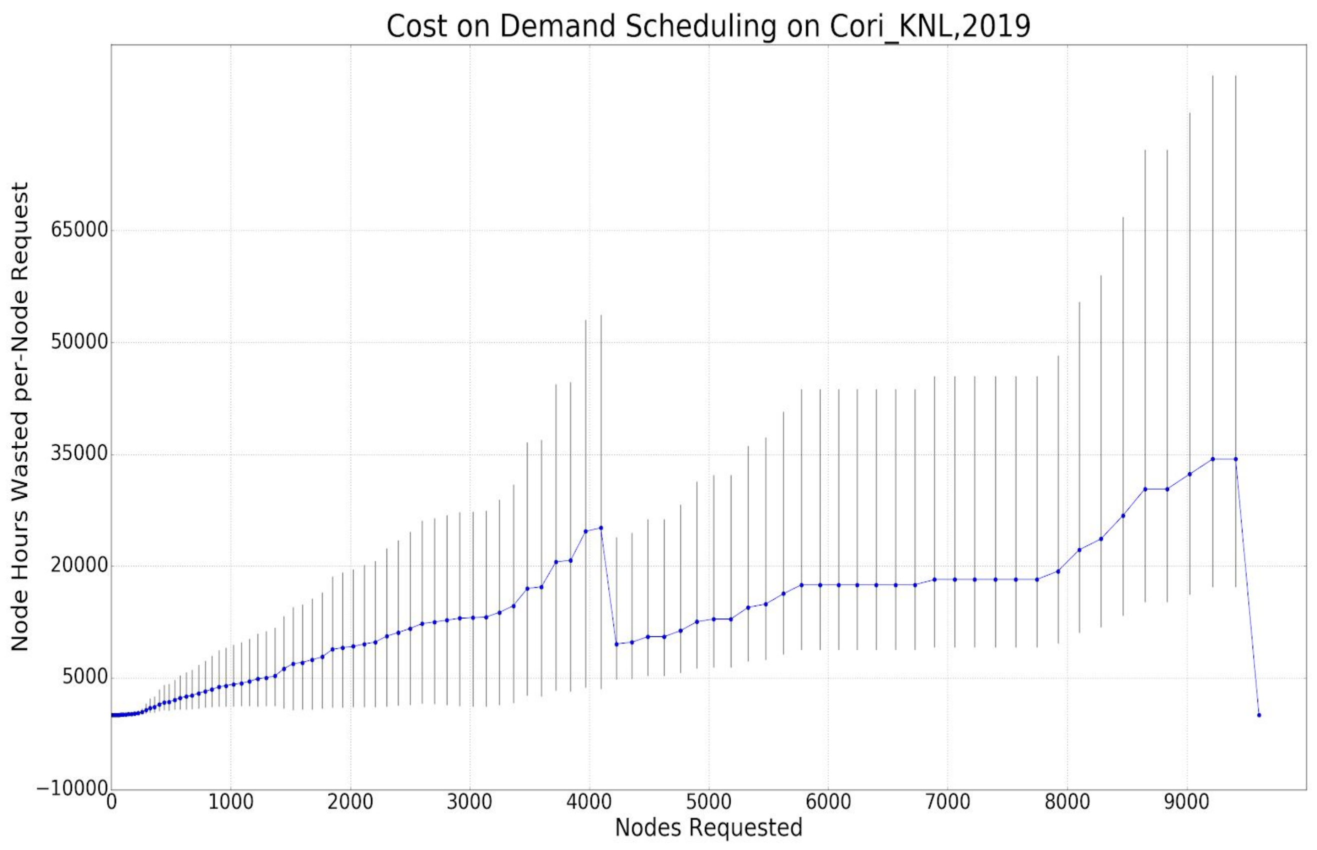

Fig. 3. Node-hours are wasted by canceling a fraction of the running batch jobs for incoming N real-time jobs.

In Fig. 3., As the number of nodes requested by the incoming real-time job increases the number of hours wasted per node request increase by canceling the running jobs per node. This can be resolved by various approaches; time threshold or time slicing, etc.

POLICY 3 - JOB PAUSING:

In this policy, we focused on measuring the impact on the system's memory when the “normal” jobs are paused and still hold the memory while the memory is allocated to the real-time job. We were able to analyse the memory pressure after the memory was given to a real-time job. The policy helps in speeding up the allocation task and it is stated as— *Upon submission of a real-time job, a fraction of the currently running batch jobs are paused (using, for example, the nice command) to allow the real-time job to start immediately. The paused jobs maintain their memory footprint, so jobs are selected for pausing to maximize the memory available to the real-time job.*

Job Pausing policy is slightly different and less expensive than Slurm's Job preemption policy.

Preemption is much more like our job cancellation policy (policy 2) which is to cancel the jobs which recently came into the queue on an arrival of real-time. It is usually used with a checkpoint restart approach. On the checkpoint-restart approach, the preempted job is expected to have written a checkpoint file in the file system while it was running. Now on the arrival of the real-time job the batch job running in the memory gets preempted, it stops immediately and all of its memory is freed to allocate the real-time job. It may or may not need to wait in the queue before it restarts. But when it restarts, the scheduler has to again read the checkpoint file from the file system and put the batch job back into the queue and not the memory. We also need to use a different library in a slurm to read and write the files into the file system in the Slurm scheduler and call the library whilst performing read-write operations at all times and loading them into the memory as well. This in addition increases the memory load. Preemption approach involves the writing of the checkpoint file in the file system and then reading it again from the file system which is really expensive.

On the other hand, the *job pausing* policy is less disruptive for the low priority user. It does not require any checkpoint file and hence it does not have to write and read the check-point file and does not have to free the memory. So in this case, when a real-time job is submitted, it pauses the batch jobs running but does not remove those jobs from the memory. This pausing can be achieved by *nice* and *renice* command. So when the batch jobs restart the scheduler does not have to read any checkpoint file from the file system and hence it does not even go through the queue process again.

The *nice* parameter is associated with Linux, a priority parameter with each job can be set or changed by the user. Linux kernel then reserves CPU time for each process based on its relative priority value. Running at a lower priority is considered "nicer", because it allows other processes to use a bigger share of CPU time.

Currently, the 'Job Pausing' policy is restricted to one real-time job per node and one paused job per node. The expected improvement through this policy is therefore quite small but still being counted. For example, if there are '**n**' real-time jobs in the queue and the '**m**' jobs paused in the memory on a node. Then all the '**m**' jobs are reserving a fraction of the memory which will eventually increase the overall waiting time of the jobs in the queue. And hence, the time of completion of the entire queue for that node will increase.

Equation 4 given below describes Memory Availability. To calculate memory available in a node at time t, $(\frac{tj}{nj} * maxrss)$ implies: $(\frac{number\ of\ tasks}{number\ of\ nodes} * maximum\ resident\ set\ size)$ which gives the total memory used by job *j* at time *t*. We are subtracting the used memory value from 96, as 96 is the maximum memory size for every single node in Cori KNL architecture.

$$Memory\ Availibility\ (t) = 96 - \sum_{j=0}' n_j * (\frac{t_j}{n_j} * maxrss_j) \quad \textbf{Eq (4)}$$

Here,

t_j	=	Number of tasks launched by job *j*.
n_j	=	Number of nodes used by job *j*.
$maxrss_j$	=	Maximum resident set size, amount of memory reserved by job *j* on runtime.

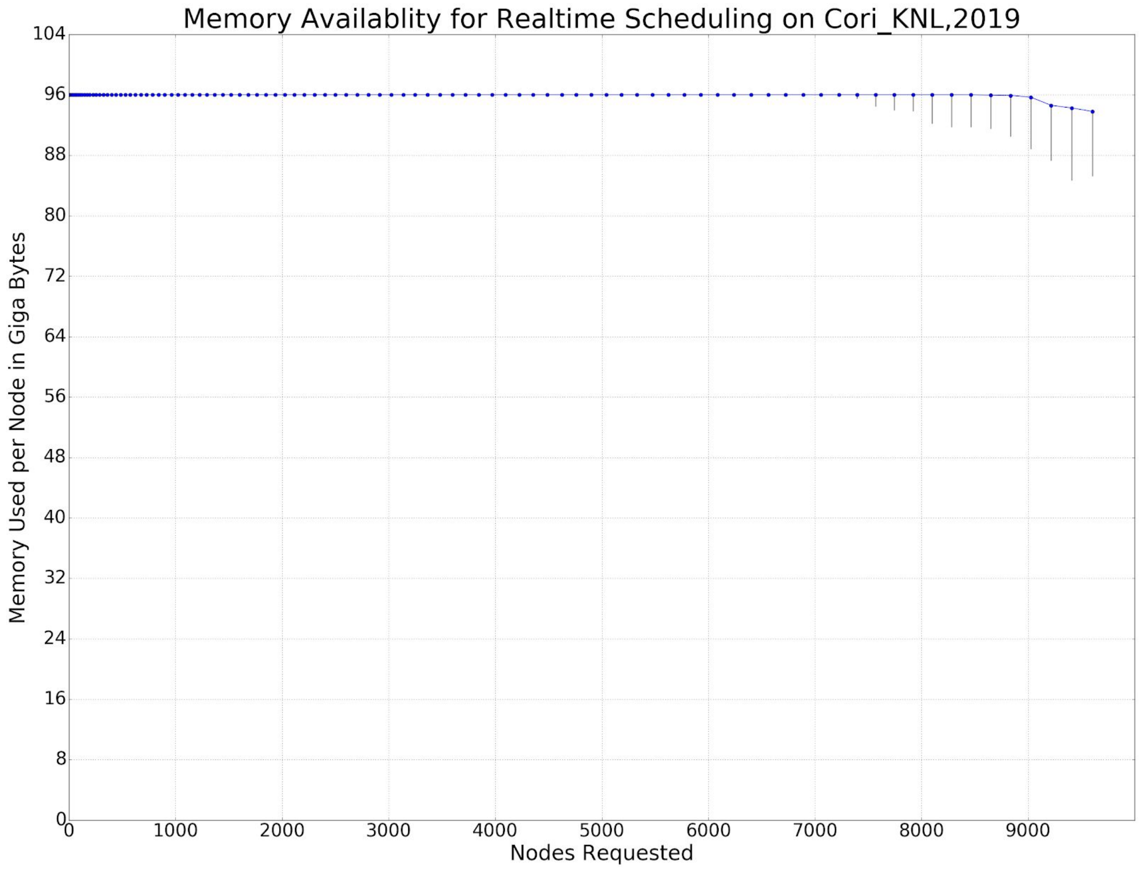

Fig. 4. Median Memory availability for the incoming real-time job on the arrival of a real-time job requiring N-nodes.

In Fig. 4. If the node request is less than 85 percent of the total number of nodes there is no memory pressure for that node for the incoming real-time job. But if in case the real-time job is really exceptional and demands more than 85 percent of the total nodes then we can see a slight increase in memory pressure.

IV. RELATED WORK

The current version of the Slurm configuration in the Cori Supercomputer is implemented by the NERSC group. The policies in the current system for real-time jobs are using backfill algorithms which are highly efficient and the overall system has approximately 98 percent system utilization, but most of the time when an arbitrary real-time job comes in, the scheduler either disrupts the other NERSC users or makes the real-time job wait under the draining process for a while before allocation in a different direction to allocate the real-time jobs. *Real-time* jobs are urgent jobs which require an immediate allocation of resources. *Premium* jobs are those users who don't want to wait in the queue and want their job done quickly. *Debug* users are users who want to access debugging instruments very frequently but do not require a large amount of nodes. The *regular* jobs are lower in priority, they are the normal jobs which wait in the regular queue for a longer time before starting. The jobs described are in table 2. below.

The current version of Cori includes usage of checkpoint-restart policy where the system has to write very big or too many checkpoint files, Cori uses burst buffers. Burst buffers are mainly used when a lot of I/O applications are involved. The file system for those checkpoint restarts is a scratch file system which is only for temporary files.

Based on table 2, we can draw the following conclusions: Real-time jobs are higher in priority than premium jobs, regular jobs and scavenger jobs. Only the percentage of nodes used are charged with the charge factor (Jobs are charged by node-hours)[8] for the shared type of jobs. QOS Debug is higher in priority but takes Max Time=0.5 hours which means it is very frequently accessed but takes a very small number of nodes. A brief outlook of a priority order is shown below.

QOS	Priority	Max Nodes	Max Time (hours)
Real-time	1	Custom	Custom
Premium	2	1772	48
Debug	3	64	0.5
Regular	4	1932	48
Shared	4	0.5	48
Scavenger	5	1772	48

Table 2. Cori Haswell - Queue Policy

The current version of Cori suffers from a lot of real-time jobs issues mostly doesn't work with the compatibility of the real-time jobs which are needed to be allocated with the resources immediately. For instance in the current scenario, when the real-time job is bigger than the node request value of 2000, the real-time job has to wait for a certain time before allocation. The previously proposed policies focused on giving more focus to real-time jobs but killing a normal job is impacting and interrupting a lot of other users' work. Consequently, Seiger Leonenkov and Sergey Zhumatiy [6] focused on developing a portable SLURM scheduler as serving many users fulfill the normal constraints but with urgent requirements meet limitations of basic SLURM schedulers. Seiger et al., discussed adding some additional features to the basic SLURM scheduler to simplify the priority system. One of the features was to bring in faster ACL checks replacing SQL based checks. However, these impacts did not show much improvement for real-time workflows for immediate allocation, but only a minute level of speed up in the overall process and improved convenience for the admin. Previous approaches, as in [9], for real-time jobs, were successful in maintaining the start of real-time within 2 minutes. Top priority jobs such as urgent jobs with smaller jobs by giving smaller jobs exclusive access to nodes. However, this technique could only achieve system usage of 92 percent and a waste of 8 percent with idle nodes.

Our work estimates the costs of these proposals using real data from the users input in the cori logs. The introduced policies maintain a better handshake and doesn't irritate the normal NERSC users nor the Real-time/Urgent job users, by not wasting much of their time.

V. FUTURE WORK

The future work involves — Machine Learning Approach. Let's divide the task into different phases. In **Phase 1**, we can collect the metric of wait-times from the draining policy graph Fig. 2.a. and also consider fetching other important details such as Job Id, Number of Nodes used by the job, Architecture used by the job, Maximum Resident Set Size, Wait-time calculated above in policy 1. In **Phase 2**, we will use a supervised machine learning algorithm (*Supervised learning is when a computer is presented with examples of inputs and their desired outputs. The goal of the computer is to learn a general formula which maps inputs to outputs.)* to train the data collected from phase 1 so as the model can understand and deduce a generalized formula. In **Phase 3**, we will use the trained supervised learning model with the new data which will be the test data to let it predict the output as wait-time based on the other configurations that user has given. With the three-phase approach we can predict the wait-time and use that information easily in Fig. 2.b. to look for idle nodes in the draining process and place the jobs which fit in that position.

Another approach could be to make decisions on normal batch jobs. That is, we can make a supervised model learn when the jobs are being canceled and what kind of jobs with what features, through the Job cancelation policy's data. And using the 3 phase approach from above we can train and predict if that job will be canceled so as not to start that batch job at that time and start it after the real-time job starts. Improvising the machine learning algorithm will impact the speed of the system and will help the Cori's Slurm scheduler make immediate decision wisely and quickly. Automating these tightly coupled and emergency decision making will be much quicker by heuristic searches (A heuristic search is a method which might not always find the best solution but is guaranteed to find a good solution in a reasonable time).

Improve these results by accounting for Backfill and predicting the running times. SLURM uses a backfilling algorithm. The running time given by the user is used as the actual running time is not known. Better running time estimations will give better performance. We can design a machine learning algorithm that; Use classic job parameters as input parameters. Works online (*to adapt new behaviors*), Use past knowledge of each user (*As each user has its own behavior*), Robust to noise i.e., *Parameters are given by humans, jobs can segfault*.

VI. CONCLUSION

Because HPC resources are limited, until now we lacked scheduling policies which can provide sufficient compute elasticity to run arbitrarily large jobs on demand. All the policies we introduced and evaluated here "pass the buck" in a different direction. That is, either the scheduler interrupts a fraction of the batch jobs for immediately allocating incoming real-time jobs or places the burden on real-time jobs by making them wait for a while before allocating resources.

System-draining delays the start of the real-time job and squanders HPC resources with idle nodes. The impact analysis shows the delays are getting bigger with the increase in node requests and could have been used by other small jobs that fit that total duration of draining. In this way, it is possible to reduce the idle-time of the Cori.

Job-cancellation gives sufficient priority to real-time jobs, but at the expense of severe disruption to other batch users whose jobs are canceled. The threshold factor discussed above will be really useful in decreasing the hours wasted by canceling jobs.

"Pausing" batch jobs is a promising option that allows real-time jobs to start immediately and avoids wasting system resources but requires more elaborate changes to the scheduler and may increase memory pressure, especially for high-concurrency real-time jobs. But memory pressure arises only when the node request value goes more than 8192, which usually has not been the case for Cori users.

By evaluating the policies of Section III, the observations show better results in terms of time and memory. Hopefully, this will be part of the implementation for the upcoming Perlmutter Supercomputer.

REFERENCES

[1] Steven Hofmeyr*, Costin Iancu, Juan A. Colmenares†, Eric Roman, Brian Austin, "Time-Sharing Redux for Large-Scale HPC Systems", *https://ieeexplore.ieee.org/document/7828392*

[2] Brian Austin, Nicholas Wright, Douglas Jacobsen, Wahid Bhimji, Tina Butler, Jack Deslippe, Scott French, Richard Gerber, "NERSC-2014 Workload Analysis", *http://portal.nersc.gov/project/mpccc/baustin/NERSC_2014_Workload_Analysis_v1.1*

[3] David Glesser, "Improve Backfilling by using Machine Learning to Predict Times in SLURM", *https://slurm.schedmd.com/SC15/SC15_BoF_Slurm.pdf*

[4] "Slurm Workload Manager", *https://slurm.schedmd.com*

[5] Rod Schultz, "Basic Configuration and Usage" *https://slurm.schedmd.com/slurm_ug_2011/Basic_Configuration_Usage.pdf*

[6] Seiger Leonenkov and Sergey Zhumatiy, "Introducing new Backfill-based scheduler for slurm resource manager", *https://www.sciencedirect.com/science/article/pii/S1877050915034249*

[7] Eashan Adhikarla, "Machine Learning Tools" *https://machinelearningtools.blogspot.com*

[8] NERSC, *https://docs.nersc.gov/jobs/policy/*

[9] Doug Jacobsen., "NERSC"*https://slurm.schedmd.com/SLUG16/NERSC.pdf*

[10] "NERSC" *https://www.nersc.gov/users/computational-systems/cori/configuration*