2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Adaptive Hybrid Queue Configuration for Supercomputer Systems

[1]Vineetha Kondameedi, [2]Sathish Vadhiyar
[1]Nvidia Graphics Private Ltd, Pune, India
[2]Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, India
vkondameedi@nvidia.com, vss@cds.iisc.ac.in

***Abstract*—Supercomputers have batch queues to which parallel jobs with specific requirements are submitted. Commercial schedulers come with various configurable parameters for the queues which can be adjusted based on the requirements of the system. The employed configuration affects both system utilization and job response times. Often times, choosing an optimal configuration with good performance is not straightforward and requires good knowledge of the system behavior to various kinds of workloads. In this paper, we propose a dynamic scheme for setting queue configurations, namely, the number of queues, partitioning of the processor space and the mapping of the queues to the processor partitions, and the processor size and execution time limits corresponding to the queues based on the historical workload patterns. We use a novel non-linear programming formulation for partitioning and mapping of nodes to the queues for homogeneous HPC systems. We also propose a novel hybrid partitioned-nonpartitioned scheme for allocating processors to the jobs submitted to the queues. Our simulation results for a supercomputer system with 35,000+ CPU cores show that our hybrid scheme gives up to 74% reduction in queue waiting times and up to 12% higher utilizations than static queue configurations.**

I. Introduction

Supercomputer centers aim to provide minimum response times for the jobs submitted to the HPC systems and maximum utilization for the systems. The HPC systems have batch queues to which jobs with specific processor size and execution time requirements are submitted. Strategies including backfilling and advanced reservation systems [1] have been proposed for the batch queues to achieve the goals of high system utilization and minimum job response times. However, the effectiveness of these strategies are constrained by the configuration of queues in the system.

Commercial schedulers for batch systems including PBS provide various configurable parameters to customize the queues of the system according to the customer's needs. Based on our studies, following are some of the essential queue configuration parameters for homogeneous high performance computing clusters:

- **Number of queues**
- **Partitioned vs Non-partitioned:** In a non-partitioned type of queuing system, all the queues share the processor space, whereas in a partitioned type of queuing system, processor space is partitioned among the queues. In a partitioned queuing system, each queue has ownership of a particular disjoint set of processors to which it can schedule its jobs for execution.
- **Request Size and Runtime Range:** Each queue can have a range for the number of processors and/or execution time duration that a job submitted to the queue can request for execution.
- **Upper limit of the queue:** The limit beyond which a queue cannot further schedule its jobs.

Based on the number of nodes allotted for running jobs: A queue can have an upper limit on the total number of processor cores used by all of its jobs that are in execution at a given point of time. In this type of queuing system, as many jobs in the queue can be scheduled for execution as long as the total number of processors used by the jobs of the queue in the execution or running state does not exceed the upper limit.
Based on the maximum allowable running jobs: A queue can have an upper limit on the number of its jobs that can be in the running state simultaneously. Production systems also employ finer controls including the maximum number of jobs per user in a queue in the running or queued state etc.

Studies of Lawson et al. [2] show that using backfilling over multiple queues results in less makespan for jobs compared to using a single queue. Multiple queues help in separating the long and short jobs to different queues based on the user estimated job execution time. This reduces the likelihood of a short job being overly delayed in the queue behind a very long job, thereby reducing the expected job slowdown. Given that using multiple queues is beneficial over having a single queue, it is non-trivial to determine the optimal number of queues to be employed and the parameters of each queue.

The system administrators may choose particular subset of configurations, based on their prior knowledge or experience, that may not be well-defined. This configuration in general, tends to be static over a period of time irrespective of the workload fluctuations, and thus cannot yield good utilization and response times under all workloads. Also, the effectiveness of the strategies such as backfilling is better realized for queue configurations that are optimal. An optimal queue configuration is a configuration that yields the minimum response time and/or maximum system utilization among all the configurations. Determining an optimal configuration for the system is not straightforward and requires good experience and knowledge of the system behavior. It is also practically not possible to test all possible configurations of the system in order to determine the best configuration.

In this paper, we have devised strategies that automatically derive a queue/system configuration for a given batch system based on the history of job submissions for reducing the

This work is supported by Department of Science and Technology (DST), India via the grant SR/S3/EECE/0095/2012.

978-1-5090-6611-7/17 $31.00 © 2017 IEEE
DOI 10.1109/CCGRID.2017.80

IEEE computer society

overall average wait time of jobs. By invoking our strategies periodically, the queue configuration can be adapted to changing workloads. In our work, we first consider/construct a partitioned type of queuing system. We propose methods to automatically determine the number of queues, partition size, request size and runtime range for each queue. We begin with a base queue configuration and determine a new queue configuration for a future period based on the workload data for the earlier period. Our process of auto tuning has two phases:

- **Phase 1 –** Determining the partition sizes for the set of queues in the previous base configuration: We have devised a novel non-linear programming (NLP) model formulation to determine the partition size for the previous set of queues based on the previous workload. In our NLP, we express the wait time of the system as a function of the processor nodes allotted to the queues, and minimize this function.
- **Phase 2 –** Determining the number of queues: Using the partitioning determined in Phase 1, we perform split and merge operations over the partitioned queues to obtain the number of queues which may be different from the number of queues in the base configuration.

We also present a novel hybrid partitioned/non-partitioned approach where we maintain a common pool of nodes which can be used by any job whose slowdown has exceeded a threshold. Our simulation results for a supercomputer system with 35,000+ CPU cores show that our proposed hybrid strategy along with dynamic queuing gives up to 74% reduction in queue waiting times and up to 12% higher utilizations than static queue configurations.

The rest of the paper is organized as follows. In Section II, we describe our queue reconfiguration methodology for partitioned system including our NLP model for partitioning, and splitting and merging algorithm to form different number of queues. In Section III, we describe our hybrid partitioned-nonpartitioned queuing strategy. Section IV presents our experiments and results. In Section V, we give details of related work. We present conclusions and give future plans in Section VI.

II. Determining Queue Configuration for Partitioned System

We first determine the partitioning of the processor space across a given set of queues, and then use these partitions to form a new set of queues. Our methodology considers as input a base queue configuration followed in the supercomputer system and workload data. The queue configuration specifies the number of queues and maximum request size and execution time duration of the jobs for each queue. The workload data is related to job submissions to the queues made in the history, namely, job parameters including the number of processors requested (request size) and the user estimated runtime, the time of submission, time of execution start and completion, and the wait time for each job of each queue. Our method assumes that the workload pattern for the prediction period will be similar to the pattern in the history.

A core component in our framework is an event driven simulator, *parsim*, we have developed to simulate a partitioned queueing system with a given queue configuration and partition sizes for the queues. *parsim* simulates submission of jobs in the workload data to the queues and their executions in the system, following specific scheduling policies. Using the simulator, we can obtain as output the start and the completion of executions of the jobs, and thus calculate waiting times and system utilization. The simulator is based on the Python Scheduler Simulator (pyss) developed by the Parallel Systems Lab in Hebrew University [3]. We configured the simulator to use the EASY backfilling algorithm [4] to schedule jobs at the individual systems.

A. An NLP-based Partitioner

The first step in our framework is to determine the partition sizes for a given set of queues in the base queue configuration. We propose a novel *NLP (Non-Linear Programming) based method* for determining the partition sizes. In our NLP method, we solve the problem of determining partition sizes as an optimization problem of distributing nodes among the queues with the objective of minimizing the overall average wait time, W, of the jobs in the system. For a given queue configuration with NQ queues, W is expressed as a function of the nodes allotted to the various queues. The objective function is then given as

$$min(W = f(P_1, P_2, P_3....P_{NQ})) \quad (1)$$

where P_i the partition size or the nodes allotted to queue i. This function is non-linear due to the inter-dependence between the nodes allotted and the wait times of the different queues. In general, as we increase the number of nodes of a given queue i, the average wait time w_i of jobs in queue i decreases. However, this increase of nodes for queue i is counter balanced by the decrease of nodes in some other queue j, which in turn will lead to increase in the average wait time w_j of queue j. This increase in w_j may be worse than the decrease in w_i which is not desirable. Figure 1 shows the effect on average wait time of the jobs with increase in the number of nodes allocated to four queues in our department's Cray cluster [1].

To determine the function, f, we use our simulator with different queue configurations with different sets of partition sizes. For each configuration with a set of partition sizes, we run the jobs submitted to the queues in the history through the simulator and obtain the average waiting times of the jobs. We thus obtain a set of average waiting times for different sets of partition sizes. We then interpolate a polynomial to fit the average wait times with the partition sizes and obtain the function, f. We used MATLAB's *polyfitn* to construct multivariate nonlinear polynomial. For simplicity, we confine ourselves to polynomials not exceeding degree 3. For our work, we ran our simulator with 100 different partition sets.

Weighted NLP: To take into account the recent trends in job submissions to the queues, we divide the history into fixed size epochs and give different weights to the different epochs in the history. For each epoch, i, we perform simulations to derive the interpolation function, f_i, for average wait time, W_i, in terms of partition sizes. We then formulate our NLP as weighted NLP as follows.

$$min(w_1 * f1 + w_2 * f2 + ... + w_i * f_i) \quad (2)$$

[1] Details of the cluster are provided in the experiment section.

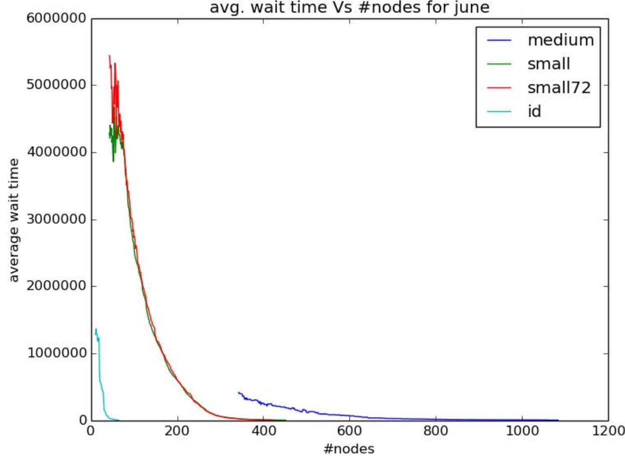


Fig. 1: Effect on waiting times with increase in number of nodes. Each curve corresponds to a different queue.

where w_i is the weight and $f_i = f(P_1, P_2, P_3....P_{NQ})$ is the interpolation function for the epoch, i. The weights to the polynomials are given based on the recency, where the most recent epoch is given more weightage.

Equality constraint: The sum of the nodes given to individual queues should be equal to the total number of nodes available in the system.

$$\sum_{i=1}^{NQ} P_i = P \tag{3}$$

where P is the total size or the total number of nodes in the system.

Lower bounds: We fix a lower bound, lb_i for the partition size for each queue i.

$$P_i \geq lb_i, \{1 \leq i \leq NQ\} \tag{4}$$

This lower bound is to disallow the NLP solver from choosing a partition size that is smaller than the largest job size submitted to the corresponding queue. Hence, one option for the lower bound is to choose the largest job size. However, this is a strict lower bound since this can result in small-sized partitions that limit the number of simultaneous job executions. Thus, the smaller the lower bound, the longer the jobs spend waiting in the queue. However, very large partition sizes can result in under-utilization of the partition and hence low CPU/system utilization. Using our simulator with the given history jobs, we simulate different partition sizes for the queues and find the relationship between system utilization and partition sizes, as shown in Figure 2 for three queues in our institute's Cray cluster. We can notice that as the number of nodes increase, the CPU utilization remains constant initially with minor fluctuations and then starts to decrease. We choose the partition size for each queue above which the utilization falls below a threshold, τ_{util}.

Proportional Allocation Constraint: We also allocate the nodes to the queues in proportion to the CPU hours consumed

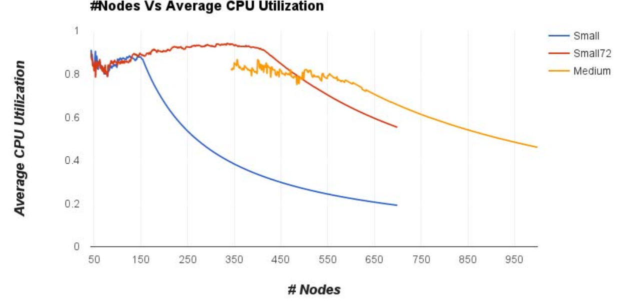


Fig. 2: Effect on CPU utilization with increase in number of nodes

by the jobs of the queues in the history. The CPU hours of a job is equal to the product of the number of processors used by the job for execution and the time taken by the job. Thus the CPU hours of a queue is given by:

$$CPUhrQ_i = \sum_{j=1}^{J_i} (CPU_{ij} \times T_{ij}) \tag{5}$$

where J_i is the number of jobs in queue i, CPU_{ij} is the number of processors used by job j submitted to queue i, and T_{ij} is the execution time of the job. We then have the following proportional allocation constraint for a pair of queues, i and j.

$$\frac{P_i}{P_j} > \frac{CPUhrQ_i}{CPUhrQ_j} \tag{6}$$

We used the function *fmincon* [5], [6] in the optimization toolbox of MATLAB to solve our non-linear optimization function. *fmincon* helps to find the local minimum of constrained nonlinear multivariate function.

B. Determining Number of Queues

After the partition sizes are determined for the queues in the given base configuration in Phase I using the NLP method, the next step is to determine the number of queues that will result in the reduction in the average response time. Thus, in Phase II, the queue configuration including the number of queues can be changed from the base configuration. We achieve this by splitting and merging the queues in the base configuration. We use the queues in the base configuration to form the new queues since the workload pattern in the history, that we use for determining the number of queues, is to an extent dictated by the base queue configuration.

1) Splitting of queues: In this method, we arrange the queues in random order or based on adjacency in terms of request size and/or runtime range in a list and perform splitting starting from the first queue of the list. After a split, we obtain a new queue configuration that we evaluate with the workload in history using our simulator. If splitting a queue results in reduced average wait time of the system, as determined by our simulator, we further split the resulting split queues until splitting gives higher average waiting time when compared to

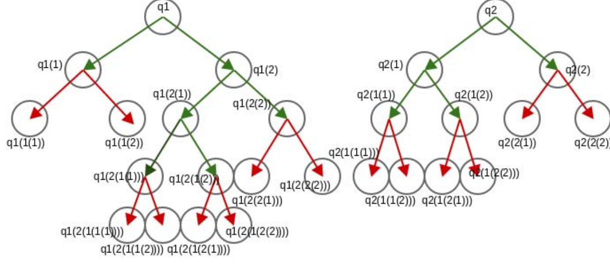


Fig. 3: Splitting of queues

the previous stage. We then visit the next queue and perform the same.

Figure 3 illustrates the splitting method for two queues q1 and q2. q1(1) and q1(2) represent first and second split sub queues of q1. The green lines represent cases where splitting has resulted in reduction of average wait times of jobs. In these cases, the method replaces the original queue from the list with two sub-queues and further split the first sub-queue. The red lines represent cases where splitting has resulted in increase of average wait times of jobs. In these cases, no further splitting of sub queues is performed, the list of queues remains the same, and the method proceeds to split the next queue in the list.

For splitting a parent queue into two subqueues, we use three methods and dynamically choose the method that gives the minimum average waiting time of the jobs. Following are the three methods.

- **Based on run-time:** In this method, we use the runtime limit of the original parent queue. We assign jobs whose runtimes are less than half of the runtime limit to the first subqueue, and the other jobs to the second subqueue. The two subqueues will have the same request size range as the parent queue, but with different runtime ranges.
- **Based on CPU hours:** The objective of this method is to create new partitions such that the resulting CPU hours of the two subqueues, formed due to splitting, is about the same. We sort the jobs of the original queue in the ascending order based on the CPUs used, i.e., the request size of the job. Starting from the job with the smallest request size in the sorted list, we keep adding the jobs to the first partition until the CPU hours of the first partition is greater than or equal to half the CPU hours of the original queue. For the remaining jobs, all the jobs having the same request size as the last job in the first partition are added to the first partition. The resulting two subqueues will have the same run time range as the parent queue, but different request size ranges as given by the ranges of the request sizes of the jobs assigned to the subqueues.
- **Based on the number of jobs:** The objective of this method is to create new partitions such that the resulting number of jobs in the two subqueues is about the same. We sort the jobs in the ascending order based on their request sizes. Starting from the job with the smallest request size in the sorted list, we keep adding the jobs to the first partition until the number of jobs in the first partition is greater than or equal to half the total number of jobs in the original parent queue. Similar to the previous method, the resulting two subqueues will have the same run time range as the parent queue, but different request size ranges.

After the jobs are partitioned into two sub queues by any of the above methods, the original nodes P_i of the parent queue are allocated proportionally to the two sub-queues based on the ratio of the CPU hours of the sub queues.

2) Merging of queues: After splitting the queues, we perform merge operation over the split queues. First, we find all possible subsets of two queues that can be formed from the original list. Next, for each subset, we check the feasibility of merging. Following are the conditions for merging two queues.

1) The request size ranges of the two queues should either be adjacent or overlapping, i.e., maximum of minimum request size limit of both queues $\leq$ minimum of maximum request size limit of both queues.
2) The runtime ranges of the two queues should either be adjacent or overlapping, i.e., maximum of minimum runtime of both queues $\leq$ minimum of maximum runtime limit of both queues.
3) Previously split queues should not be merged, i.e., the parent of the two queues must not be the same.

Conditions 1 and 2 are employed to avoid the formation of any redundant queues due to the merging of two queues whose request size and/or runtime ranges are disjoint. If any of the above condition is not satisfied for a subset, the subset is discarded. For the remaining feasible subsets, we perform merge operations individually on each of them, evaluate the new configuration using our simulator, and select the subset having the minimum average wait time. If the average wait time of jobs of the new configuration is less than that obtained from the previous configuration, we merge both the queues in that subset, and replace the two queues in the original list with the new merged queue. The new configuration will now act as a parent configuration in the next iteration. We repeat the above process until there is no feasible subset for merging or no new configuration's average wait time is less than the previously achieved average wait time.

III. Hybrid Partitioned-NonPartitioned Strategy

Our methods of NLP, splitting and merging result in a partitioned queue system in which the processor space is partitioned among the queues. Partitioned system can result in low utilization if the workloads of the different queues are skewed and the queues corresponding to some of the partitions have low workloads. Figure 4 shows the workloads in the form of CPU hours of three queues in our institute's Cray cluster. As can be seen, there are instances when the workload of one queue are much higher than the workload of another queue. In a partitioned system, each queue has ownership of disjoint

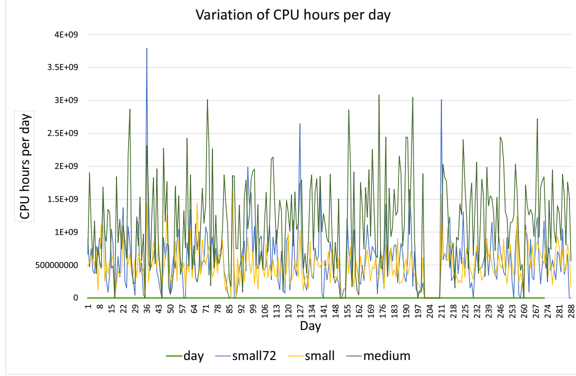


Fig. 4: CPU hours per day over the period of nine months

set of nodes which can only be used to schedule the jobs of that queue. Having rigid boundaries will have adverse affects when the workload is skewed. While in one queue, the queue length increases with incoming jobs, the nodes in the other queue are under-utilized. On the other hand, a non-partitioned system can result in starvation of jobs with certain ranges of request sizes and runtimes. For example, jobs of large queues with large request sizes can fill up the processor space, starving the smaller jobs of processors.

In order to overcome the challenges due to such skewed workloads, we propose a hybrid partitioned/non-partitioned model in which a part of the processor space is partitioned among the queues and the remaining processor space forms a common pool for use by jobs of all the queues. We begin with a partitioned system and carve out processors from the partitions to form a common pool of processors. We traverse the queues in the increasing order of partition sizes in a round-robin manner. In each round for each queue, we remove processors from its partition for adding to the common pool. We remove a fixed chunk of size $chunkSize$ of processors from the partition and evaluate the new partitioned queue configuration using our simulator. Reducing the partition size can increase the average waiting time of the jobs in the queue. If the average wait time of the jobs in the queue with the reduced partition does not exceed the average wait time for the queue with the original partition by more than a degradation percentage, $degPercent$, we add the chunk to the common pool. Else, we stop shrinking the partition for the queue any further. We then move to the next queue. In this way, we keep adding the chunks to common pool from the queues until no queue can further be shrunk or until the overall average wait time of all the jobs in all the queues itself has exceeded $degPercent$.

The nodes from the common pool can be used by a job submitted to a queue based on a *priority factor* of the job. The priority factor considers both the current slowdown of the job and the priority of the queue.

$$slowdown = \frac{(current_time - submit_time)}{min(est_run_time, queue_max_wallTime)}$$

$$priorityfactor = (slowdown) * (priority\ of\ queue)$$

If the priority factor is greater than a threshold, τ_{pf} and if sufficient nodes are not available in the queue to schedule the job, then the scheduler tries to schedule the job using the nodes of the common pool.

Fairness to Jobs: Our techniques provide fairness to jobs in a number of ways. We set starvation time for each queue, exceeding which a job is given higher priority. If any queue has starving jobs, the nodes allocated to the queue are not considered for scheduling of large jobs in the partitioned system. Further, in the case of hybrid system, nodes of the common pool are allocated based on the priority factor, as mentioned above.

Practicality of our Solution: Our method is intended for the supercomputer installations that are willing to adapt the queue configurations of their local queue managers like PBS based on the outputs from our methods. In the worst case, this will involve draining out the system of all the current jobs and restarting the job manager with the new queue configuration.

IV. Experiments and Results

A. Experiment Setup

We performed simulations using the queues and workload data of our institute's Cray XC40 cluster located in and maintained by Supercomputer Education and Research Centre (SERC). It has three kinds of nodes:

- 1468 CPU-only nodes with each node consisting of dual Intel Xeon E5-2680 v3 (Haswell) twelve-core processor at 2.5 GHz for a total of 35232 CPU cores,
- 48 Intel Xeon Phi nodes with each node consisting of an Intel Xeon E5-2695 v2 (Ivybridge) twelve-core processor CPU at 2.4 GHz and an Intel Xeon Phi 5120D for a total of 576 CPU cores and 50 Xeon Phis, and
- 44 GPU nodes with each node consisting of an Intel Xeon E5-2695 v2 (Ivybridge) twelve-core processor CPU at 2.4 GHz and an NVIDIA Kepler K40 GPU for a total of 528 CPU cores and 50 GPUs.

The nodes are connected by Cray Aries interconnect using DragonFly topology.

The Cray system follows non-partitioned type of queuing system with seven queues as shown in Table I. As shown in the table, the system has five CPU-only and two accelerator queues. We confine our experiments to the CPU-only queues. The queues employ aggressive backfilling and starvation with starvation period of 24 hours. The upper limits of each queue are defined by the maximum allowable running jobs from the queue.

Table II and Figure 5 summarize the workload data of the Cray system over a period of nine months considered in our experiments. The table shows the workload in terms of the number of jobs submitted to the queues and the figure shows the CPU hours of the jobs in the various queues. As shown in the figure, the workload variation is drastic for two instances: from November to December 2015, and from January to February 2016.

Queue	min cpus	max cpus	min wall time	max wall time
idqueue	8	256	NA	2:00:00
small	24	1032	NA	24:00:00
small72	24	1032	24:00:00	72:00:00
medium	1033	8208	NA	24:00:00
large	8209	22800	NA	24:00:00
gpu	1	60	NA	24:00:00
xphi	1	60	NA	24:00:00

TABLE I: Queue Configurations in the Cray System

month	idqueue	small	small72	medium	large
July'15	3013	1066	238	324	19
Aug'15	4205	1171	245	1101	13
Sep'15	4161	978	207	454	10
Oct'15	2917	1441	244	640	19
Nov'15	2996	1122	290	454	13
Dec'15	2162	943	179	332	4
Jan'16	2938	844	191	352	2
Feb'16	5198	1410	233	433	8
March'16	6449	1588	216	503	3

TABLE II: Workload in the Cray System: Number of Jobs

The simulation experiments were performed using our *parSim* simulator. Inputs to parSim are queue manager file and system logs. It initializes queues of the system and system scheduling policies based on the parameters in the queue manager file. For partitioned system with the CPU-only queues of the Cray system, the sizes of the partitions for the queues should be at least equal to the maximum request sizes for the queues. However, as shown in Table I, the maximum request size for the *large* queue is 22,800 cores. Allocating a large partition of size at least equal to 22,800 cores is not desirable since that leaves small number of processors for the jobs of the other queues and will also lead to inefficient system utilization due to small number of large jobs. Hence for jobs of the *large* queue, the partitioned system employs advanced reservations.

Our methodology employs a number of parameters. One of the important parameters is the *history period* after which the queues are considered for reconfiguration. Unlike continuous learning and corrections employed in many applications of machine learning, in HPC systems it is practically not possible for the system administrators to reconfigure the queues frequently since reconfiguration involves flushing out the jobs in the existing queues, a period of potential shutdown of the system, and advanced notifications to the users. As shown in Figure 4, the workload in the Cray system queues show high variations from one day to another. Thus, a day's workload cannot be reliably used as the history period. Figures 6(a) and 6(b) show the workload variations for intervals of 10 days and for every month, respectively. We find that similar to per day interval, the workload shows highly varying non-deterministic pattern for ten-day intervals. However, as we aggregate the data over the period of one month, we can observe that the noise is significantly reduced and the pattern is also noticeable. This suggests that for the Cray system, we can assume that the workload data of a previous month is a reliable indicator for the workload in the upcoming month. Hence our methods use workload data for a month as the history period, and reconfigure the queues for the subsequent month's usage. Such monthly reconfiguration of queues is also practical and advisable in some supercomputer systems. We performed such sensitivity studies and set the values for the other parameters as shown in Table III.

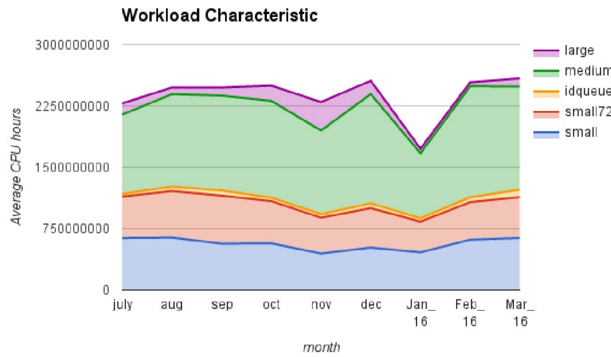


Fig. 5: Workload in the Cray System: Average CPU hours of each queue

Parameter	Value
Weighted NLP:	
Number of epochs of the previous one-month history	3 10-day epochs
Weights for the epochs (in chronological order)	1/3, 2/3, 3/3
τ_{util} threshold for lower bound constraint	90% of max utilization
Hybrid Strategy:	
$chunkSize$	5 processors
Degradation percentage, $degPercent$	30%
τ_{util} threshold for priority factor	0.5*priority of the *medium* queue

TABLE III: Parameters used in our method

We evaluate and compare the following strategies for queue configurations:

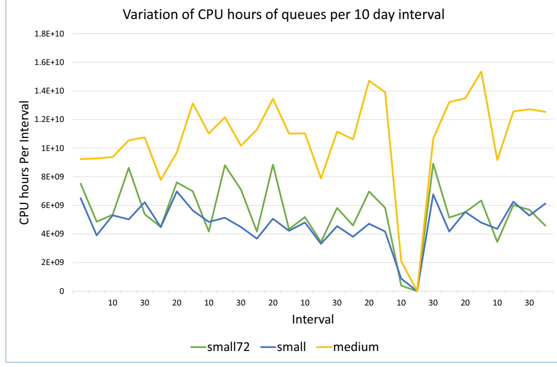
1) **Proportional Partitioning (PP):** This method partitions the processor nodes proportionally among the queues based on the fraction of CPU hours of the jobs in each queue in the history. Thus, the partition size for a queue, P_q, is defined as:

$$P_q = \left(\frac{\sum_{j=1}^{J_q}(CPU_{qj} \times T_{qj}}{\sum_{i=1}^{NQ}(\sum_{j=1}^{J_i} CPU_{ij} * T_{ij})}\right) \times P \qquad (7)$$

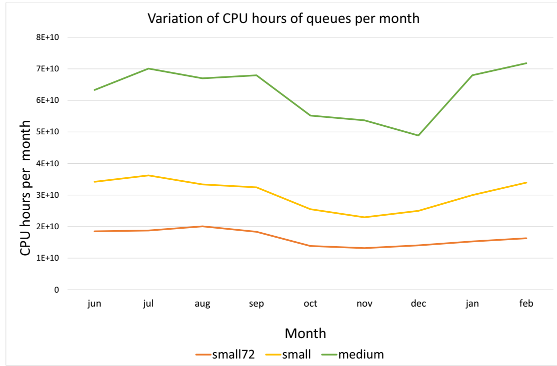
where P is the total number of processors, NQ is the number of queues, J_q is the number of jobs in q, CPU_{ij} is the number of processors used by job j submitted to queue i, and T_{ij} is the execution time of the job j submitted to queue i. While this simple method considers the dynamic workloads, it does not use the arrival rates and the wait times of the jobs in the workload.

2) **Static Configuration:** This method follows one queue configuration, that is fixed initially, for all the months. For fixing the initial configuration, we use the output of proportional partitioning for the first month of simulation.

3) **Dynamic partitioning (DP-NLPM):** Here, the number of queues and queue parameters including the run time limits/request size limits etc are maintained the



(a) CPU hours per interval of ten days



(b) CPU hours per month

Fig. 6: Workload Variations for a nine-month period in the Cray System

same as the original configuration. Only the partition size of each queue is varied based on the output of our NLP method.

4) **Dynamic queue reconfiguration (DQR):** In this method, the queue configuration including the number of queues are changed using the split and merge operations over the dynamically partitioned queues.

5) **Hybrid System:** This is the hybrid partitioned/non-partitioned queuing system, using our method, containing partitioned queues along with a common pool of nodes.

B. Results

1) Comparison of Dynamic Methods with Static Configuration: Figure 7 shows comparative analysis of dynamic methods with respect to the static configuration in terms of average waiting times of the jobs. The dynamic partitioning methods, namely, proportional partitioning (PP), dynamic partitioning (DP-NLPM) and dynamic queue reconfiguration (DQR) give better performance than the static configuration in almost all

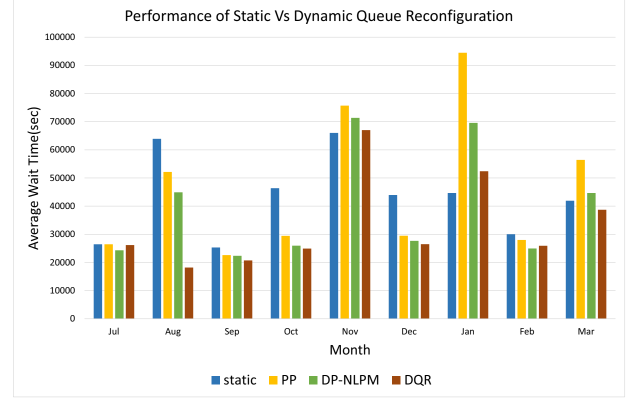


Fig. 7: Comparison of Dynamic Strategies over Static Configuration in terms of Average Waiting Times

the cases except for a few months, especially for January'16. Leaving the results for January, the DP-NLPM method gives average reduction of 16% and maximum reduction of 44% in the average waiting time of jobs when compared to the static configuration. The DP-NLPM method performs better than the proportional partitioning (PP) method for all the months. The DP-NLPM method gives average reduction of 12% and maximum reduction of 26% in the the average waiting time of jobs when compared to the PP method. The reason for this is that unlike proportional partitioning where we have tight bounds, the constraints in NLPM are not rigidly fixed. The optimizer has a larger search space to find an optimal solution. Also, the weighted NLPM gives more weightage to the last week of the month than the first week. There is higher probability that the pattern followed in the last week is repeated in the next month.

Dynamic queue reconfiguration (DQR) further reduces the average waiting time by 12% average and 59% maximum reduction when compared to DP-NLPM, and performs the best in almost all the months. The method, by performing splitting and merging over the queues, reconfigures the whole queue configuration to suit the underlying workload distribution. This suggests that it is advisable to even change the number of queues based on the workloads.

Figure 8 shows the system utilization due the different methods. The utilization values are plotted between 0 to 1, with 1 representing 100% utilization. We find that our dynamic methods not only gives reduced average wait times, but also higher utilizations than the static configuration, thus providing fairness to both short (small wait times) and long running jobs (high utilization). The dynamic queue reconfiguration method with change in the number of queues gives the highest utilization values in most cases.

The lower performance of our dynamic methods for January'16 is due to drastic reduction in CPU hours of medium queue from the previous month which can be noticed in Figure 5. Considering the previous month's workload, the

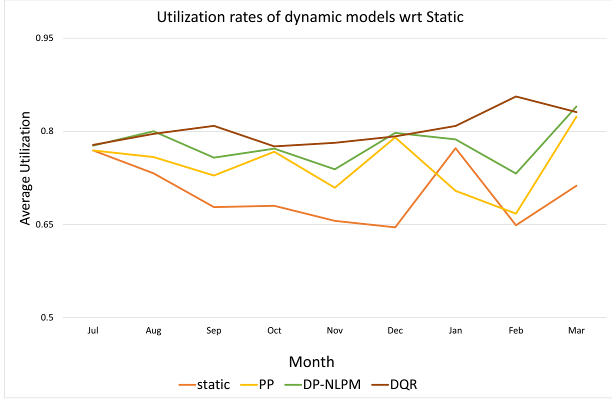


Fig. 8: Comparison of Dynamic Strategies over Static Configuration in terms of System Utilization

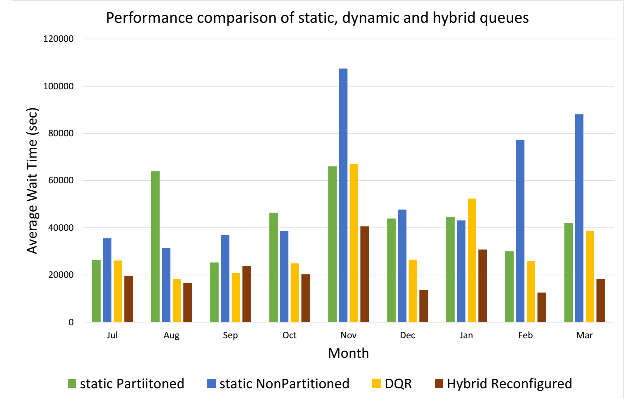


Fig. 9: Comparison of Hybrid Strategy over Static Configurations in terms of Average Waiting Times

model allocated more nodes to medium queue when compared to the other queues. A closer look at the percentage of nodes allocated show that for January'16, the proportional partitioning method allocated 57% of the nodes to medium, 22% of the nodes to small and 21% of the nodes to small72. So, while the jobs in small and small72 queues did not have sufficient nodes for scheduling, the nodes in the medium queue were being under utilized. This motivates the use of our hybrid strategy, where we maintain a common pool of nodes, and nodes in the common pool can be considered for scheduling a job that undergoes starvation.

2) Hybrid Model: In this section, we compare our proposed hybrid partitioned/non-partitioned model with respect to static partitioned queuing system and static non-partitioned queuing system. The Cray XC40 system follows non-partitioned type of queuing system. Hence, the non-partitioned system configurations we have considered are of Cray XC40.

Figure 9 compares the performance of hybrid strategy with the static queuing systems in terms of average waiting times. The hybrid partitioned/non-partitioned queuing strategy outperforms the static queuing in all the cases. When compared to the static configuration, the hybrid system gives average reduction of 46% and maximum reduction of 74% in average waiting times. Note that the partitioned system using dynamic queue reconfiguration (DQR) gave only an average reduction of 24% over the static configuration, as shown earlier in Figure 7. This suggests that the hybrid strategy, which uses the partitions created from the DQR method to form a hybrid system, improves the performance of the dynamic partitioned system. When compared to the non-partitioned queuing system, the hybrid model gives average reduction of 55% and maximum reduction of 83%.

Though reduction in average wait time of jobs was our primary objective, bounded slowdown is another metric which is widely used for evaluations in job scheduling. It is defined as $(waitTime + max(runTime, threshold))/max(runTime, threshold)$.

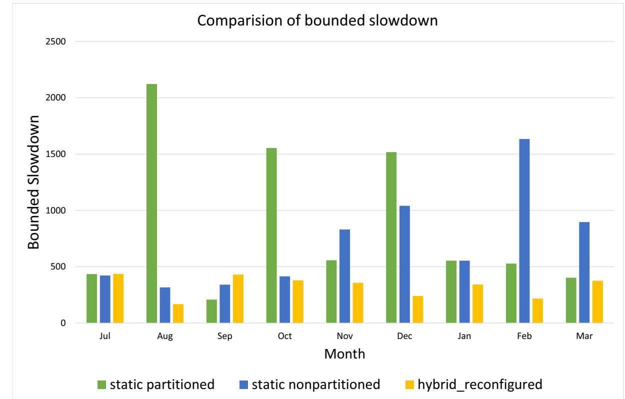


Fig. 10: Comparison of Hybrid Strategy over Static Configurations in terms of Average Bounded Slowdown

It is one of the metrics that analyze fairness: jobs with smaller run times should have smaller wait times for the bounded slowdown values to be small. Figure 10 shows the comparative results of the bounded slowdown using the three configurations, namely, static (partitioned), static (non-partitioned)/Cray and hybrid_reconfigured queue configurations. Our proposed hybrid configuration outperforms the static configurations in almost all the months. On an average, the percentage reduction of bounded slowdown of the hybrid system in comparison to the static partitioned system is around 39% and in comparison to static non-partitioned system is around 35%.

Figure 11 shows the utilization of the system using the hybrid and the static methods. System utilization in hybrid reconfigured queues is on an average 12% more than the static partitioned queuing. In case of non-partitioned type of queuing

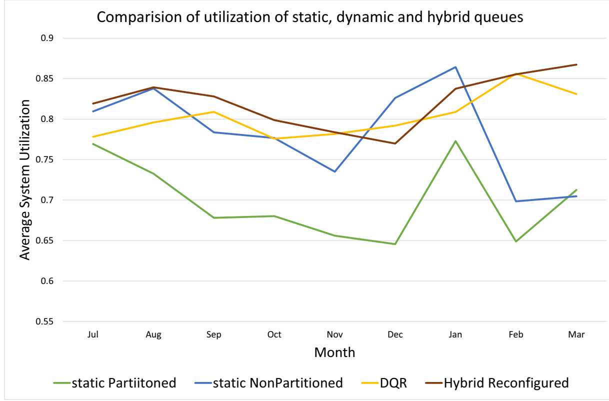


Fig. 11: Comparison of Hybrid Strategy over Static Configurations in terms of Utilization

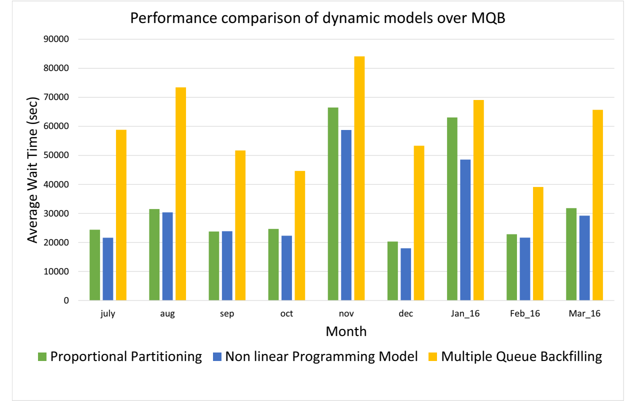


Fig. 12: Comparison with Existing Work in terms of Average Waiting Times

system, though the average wait time experienced by the jobs is higher than that of the hybrid system, the utilization in some cases is comparable or greater than that of the hybrid queuing. This is because of the contradictory nature of utilization and average response time of the system. The primary reduction in overall average wait time of the jobs in the system is due to lower response times for small jobs, while the system utilization improves due to long and large jobs. Our methods aim to strike a balance between these two factors. In the Cray system, high priority is given to large jobs when compared to small jobs and hence results in higher utilization in some cases.

Comparing these utilization results of the hybrid system with the utilization results of the partitioned system due to our dynamic queue reconfiguration (DQR), shown in Figure 8, we find that the utilization in the hybrid system is 3-4% higher than the utilization in the DQR system in most cases. As noted earlier, the hybrid system gives reduced waiting times when compared to the DQR system. Thus, the hybrid system provides both improved waiting times and utilization when compared to the corresponding partitioned system. The waiting time improvement is because the jobs with slowdowns beyond a threshold are allocated nodes from the common pool, and improved utilization is because the hybrid system, by reducing the partitions for creating the common pool, does not cause under-utilization of queues for skewed workloads unlike the corresponding partitioned system.

3) Comparison with Existing Work: We also compared our work with the existing work on multiple queue backfilling scheduling by Lawson and Smirni. [7], [8] where they maintain flexible partition boundaries[2]. Figure 12 shows that the average waiting time due to our DP-NLPM approach is smaller than that of the multiple queue backfilling by Lawson and Smirni (marked as “MQB”) by an average of 50%. The main reason for such poor performance of their work is because their multiple queue backfilling scheduling hugely depends on the backfilling strategy and is adversely affected if the user run time estimates are inaccurate. In the workload data which we have used, the run time estimates in most cases are equal to the maximum wall time of queues.

[2]Details given in Related Work section

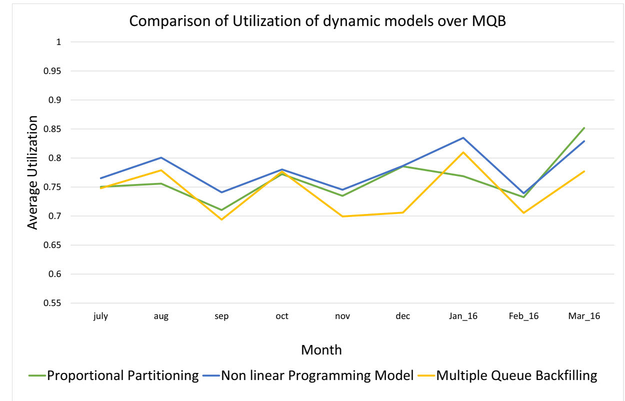


Fig. 13: Comparison with Existing Work in terms of Utilization

Figure 13 compares the utilizations of our work and the multiple queue backfilling (MQB) method. We find that our hybrid method gives up to 8% higher utilization than the MQB method.

V. Related Work

Prabhakaran et al. [9] distinguishes the dynamic and static requests and introduces new fairness schemes for improved resource allocation. VARQ [10] (Virtual Advanced Reservations for Queues) is a system that uses QBETS [1] to implement an advanced reservation abstraction for HPC users. QBETS computes a time bound on the delay a specific user job will

experience. The partitioned system simulator that we present, is based on the advanced reservations for large jobs. Our work differs from all the above work by focusing on adaptive queue reconfigurations.

Krishnamurthy et al. [11] provided a toolset that can help a system administrator to automatically configure a scheduling policy. Streit [12], [13] has proposed a self-tuning *dynP* job scheduler which can tune queuing policy dynamically during run time. Tang et al. [14] propose metric-aware scheduling which enables the scheduler to balance scheduling goals represented by different metrics.

Lawson and Smirni [7] proposed multiple queue backfilling strategies with priorities and reservations for parallel systems. Their work emphasizes on dynamically changing the partition size of the queue for effective backfilling of any of the queued job, without delaying the high priority job of other partitions. In their subsequent work [8], they proposed method to automatically change the number of partitions on-the-fly, to address transient workload fluctuations. Their work focused on deriving optimal number of partitions once the slowdown exceeds a predefined threshold. In their work, they perform online simulation starting from one partition to P_{max} (maximum number of allowable partitions) partitions of the system using the currently queued jobs in the system. Finally the configuration which gives best overall performance is adopted.

One of the major drawbacks of their work is that they perform simulations on-the-fly and change the system configuration every time the slowdown exceeds a predefined threshold τ. Reconfiguring the queues on-the-fly has high overheads since it involves waiting for all the queues to be drained before reconfiguration, which also reduces the system utilization. However, our work differs from their work in several aspects. Instead of changing the number of partitions on the fly, we consider a window period of one month. Based on the observations of previous month workload data and system behavior, we determine optimal system configuration. Unlike their work [8], we do not perform our simulations starting from one queue upto P_{max}. Rather, we perform reconfigurations over the previous partitions. This is because the workload pattern in the previous month is influenced by the system configuration of the month.

VI. Conclusions and Future Work

Most of the super computing centers face the problem of high wait time to the users and low utilization of the the system. Our results obtained show that using a fixed configuration affects both performance and utilization of the system adversely. We have proposed a method based on NLP to dynamically reconfigure the queue partition size based on the workload of previous months. Our NLP-based method (DP-NLP) gives up to 44% reduction in queue waiting times when compared to static queue configurations. Using the partitioned queues we perform split and merge operations over the queues to determine the optimal number of queues. This dynamic reconfiguration of queues (DQR) result in up to 59% reduction in queue waiting times and higher utilizations over the partitioned system obtained using DP-NLP. We have shown the cases of non-determinism in the workloads and how this can adversely affect the performance of the system. In order to overcome the non-deterministic nature of the workload, we have come up with a hybrid partitioned/non partitioned way of queuing, where a common pool of nodes are maintained apart from the normal partitioned queues. These common pool of nodes can be used by jobs if their slowdown exceeds a threshold. Our hybrid system gave further reductions in queue waiting times and slowdowns, and higher utilizations over the partitioned system due to the DQR method.

Currently, the parameters which we consider for auto tuning are the number of queues, queue partition size, queue priority, queue run time limits, and queue CPU limits. Commercial Maui schedulers have nearly 200 configurable parameters. In future, we would like to develop a generic tool that works on any type of queuing system, partitioned and Non-partitioned. The tool will help reduce the difficult task of finding optimal configuration of the system and help the HPC communities reduce the waiting time of the jobs.

References

[1] D. Nurmi, J. Brevik, and R. Wolski, "Qbets: queue bounds estimation from time series," in *Job Scheduling Strategies for Parallel Processing*. Springer, 2007, pp. 76–101.

[2] B. G. Lawson, E. Smirni, and D. Puiu, "Self-adapting backfilling scheduling for parallel systems," in *Parallel Processing, 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 583–592.

[3] O. Peleg, "Python Scheduler Simulator," Feb. 2010. [Online]. Available: http://code.google.com/p/pyss/

[4] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, "Parallel Job Scheduling: a Status Report," in *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, ser. JSSPP'04, 2005, pp. 1–16.

[5] "Constrained nonlinear optimization algorithms," http://in.mathworks.com/help/optim/ug/constrained-nonlinear-optimization-algorithms.html.

[6] "http://bwrcs.eecs.berkeley.edu/Classes/icdesign/ee141_s03/Project/Project1_solutions/fmincon.pdf".

[7] B. G. Lawson and E. Smirni, "Multiple-queue backfilling scheduling with priorities and reservations for parallel systems," in *Job Scheduling Strategies for Parallel Processing*. Springer, 2002, pp. 72–87.

[8] B. Lawson and E. Smirni, "Self-adaptive scheduler parameterization via online simulation," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*. IEEE, 2005, pp. 29a–29a.

[9] S. Prabhakaran, M. Iqbal, S. Rinke, C. Windisch, and F. Wolf, "A batch system with fair scheduling for evolving applications," in *Parallel Processing (ICPP), 2014 43rd International Conference on*. IEEE, 2014, pp. 351–360.

[10] D. Nurmi, R. Wolski, and J. Brevik, "Probabilistic reservation services for large-scale batch-scheduled systems," *Systems Journal, IEEE*, vol. 3, no. 1, pp. 6–24, 2009.

[11] D. Krishnamurthy, M. Alemzadeh, and M. Moussavi, "Towards automated hpc scheduler configuration tuning," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 15, pp. 1723–1748, 2011.

[12] A. Streit, "Enhancements to the decision process of the self-tuning dynp scheduler," in *Job Scheduling Strategies for Parallel Processing*. Springer, 2004, pp. 63–80.

[13] ——, "Evaluation of an unfair decider mechanism for the self-tuning dynp job scheduler," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 108.

[14] W. Tang, D. Ren, Z. Lan, and N. Desai, "Adaptive metric-aware job scheduling for production supercomputers," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. IEEE, 2012, pp. 107–115.