Prediction of Queue Waiting Times for Metascheduling on Parallel Batch Systems

Rajath Kumar and Sathish Vadhiyar

Supercomputer Education and Research Center, Indian Institute of Science, Bangalore, India
{rajath.kumar@gmail.com,vss@serc.iisc.in}

Abstract. Prediction of queue waiting times of jobs submitted to production parallel batch systems is important to provide overall estimates to users and can also help meta-schedulers make scheduling decisions. In this work, we have developed a framework for predicting ranges of queue waiting times for jobs by employing multi-class classification of similar jobs in history. Our hierarchical prediction strategy first predicts the point wait time of a job using dynamic k-Nearest Neighbor (kNN) method. It then performs a multi-class classification using Support Vector Machines (SVMs) among all the classes of the jobs. The probabilities given by the SVM for the class predicted using k-NN and its neighboring classes are used to provide a set of ranges of predicted wait times with probabilities. We have used these predictions and probabilities in a meta-scheduling strategy that distributes jobs to different queues/sites in a multi-queue/grid environment for minimizing wait times of the jobs. Experiments with different production supercomputer job traces show that our prediction strategies can give correct predictions for about 77-87% of the jobs, and also result in about 12% improved accuracy when compared to the next best existing method. Experiments with our meta-scheduling strategy using different production and synthetic job traces for various system sizes, partitioning schemes and different workloads, show that the meta-scheduling strategy gives much improved performance when compared to existing scheduling policies by reducing the overall average queue waiting times of the jobs by about 47%.

1 Introduction

Production parallel systems in many supercomputing sites are batch systems that provide space sharing of available processors among multiple parallel applications or jobs. Well known parallel job scheduling frameworks including IBM Loadleveler [1] and PBS [2] are used in production supercomputers for management of jobs in the batch systems. These frameworks employ batch queues in which the jobs submitted to the batch systems are queued before allocation by a batch scheduler to a set of available processors for execution. Thus, in addition to the time taken for execution, a job submitted to a batch queue incurs time due to waiting in the queue before allocation to a set of processors for execution.

Predicting queue waiting times of the jobs on the batch systems will be highly beneficial for users. The predictions can be used by a user for various purposes including planning management of his jobs and meeting deadlines, considering migrating to other queues, systems or sites at his disposal for application execution when informed of possible high queue waiting times on a queue, and investigating alternate job parameters including different requested number of processors and estimated execution times. Such predictions can also be efficiently used by a meta-scheduler to make automatic scheduling decisions for selecting the appropriate number of processors and queues for job execution to optimize certain cost metrics, and help reduce the complexities associated with job submissions for the users. The decisions by the user and meta-scheduler using the predictions can in turn result in overall load balancing of jobs across multiple queues and systems. Such predictions are also highly sought after in production batch systems. For example, predictions of queue waiting times are available in production systems of TeraGrid [3]. These show the importance of accurate queue wait time prediction mechanisms for the users submitting their jobs to batch systems.

Prediction of queue waiting times is challenging due to various factors including diverse scheduling algorithms followed by the job scheduling frameworks, time-varying policies applied for a single queue, and priorities for the jobs. In our previous work [4], we have developed a framework called PQStar (Predicting Quick Starters) for identification and prediction of jobs with short actual queue waiting times. We refer to these jobs as *quick starters*. The basis of our method for identifying a quick starter job is to establish boundaries in the history of prior job submissions, and to use the similar jobs within the boundaries for prediction. Thus, the most relevant and recent history is used for predicting the target quick starter job. For our work, we defined quick starters as those jobs with actual waiting times of less than one hour, since these formed the majority in many real supercomputing traces. The prediction strategies lead to correct identification of up to 20 times more quick starters and resulted in up to 64% higher overall prediction accuracy than existing methods.

In this work, we extend our framework with strategies for predicting ranges of queue waiting times for all classes of jobs, and use these stochastic predictions to build a top-level meta-scheduler to reduce the overall average wait times of the jobs submitted to the queues. We have developed a machine learning based framework that identifies for a target job, similar jobs in history using job characteristics and system states, and employs multi-class classification method to provide predictions for the jobs. Our hierarchical prediction strategies first predict the point wait time of a job using the *Dynamic-k Nearest Neighbor* (kNN). It then performs a multi-class classification using *Support Vector Machines (SVMs)* among all the classes of the jobs. The probabilities given by the SVM for the predicted class obtained from the kNN, along with its neighboring classes, are used to provide a set of ranges of wait times with probabilities. An important aspect of our prediction model is that it considers the processor occupancy state and the queue state at the time of the job submission in addition to the job

characteristics including the requested number of processors and the estimated runtime. The processor and queue states include the current number of free nodes, number of jobs with large request sizes currently executing in the system, and relative difference between the current job and other jobs in the queue in terms of request size and estimated run time. These states are obtained by using a simulator that updates the states during job arrivals and departures.

We have also developed a meta-scheduling framework which uses the predictions of queue waiting times to select a queue for job submission and execution. The primary objective of our meta-scheduler is to distribute jobs to different queues/sites in a multi-queue/grid environment for minimizing average wait times of the jobs submitted to the queues. For a given target job, we first identify the queues/sites where the job can be a quick starter to obtain a set of candidate queues/sites and then compute the expected value of the wait time in each of the candidate queues/sites using the ranges of queue waiting times and probabilities. Our meta-scheduler then schedules the job to the queue with minimum expected value for job execution.

Our experiments with different production supercomputer job traces show that our prediction strategies can give correct predictions for about 77-87% of the jobs, and also result in about 12% improved accuracy when compared to the next best existing method. Our experiments with our meta-scheduling strategy using different production and synthetic job traces for various system sizes, partitioning schemes and different workloads, show that our meta-scheduling strategy gives much improved performance when compared to existing scheduling policies by reducing the overall average queue waiting times of the jobs by about 47%.

2 Related Work

In the works by Smith et al. [5] [6], runtime predictions are derived using similar runs in the history, and these estimates are further used to simulate the scheduling algorithms like FCFS, LWF (Least Work First) and Backfilling [7] to obtain the queue wait times predictions. Some statistical methods use time series analysis of queue waiting times for jobs in the history to predict waiting times for submitted jobs. QBETS [8, 9] is a system that predicts the bounds on the queue wait times with a quantitative confidence level. However, QBETS gives conservative upper bound predictions, which leads to large prediction errors for most of the jobs. Also it considers only the job characteristics and not the state of the system.

The efforts by Li et al. [10,11] consider the system states for the prediction of queue waiting times. In their method known as Instance Based Learning (IBL), they use weighted sum of Heterogeneous Euclidean-Overlap Distances between different attributes of two jobs to find the similarities between the jobs. Their work gives only point predictions for wait times while we provide ranges of wait times with probabilities which we show as more useful than the point predictions.

One common strategy for scheduling jobs in a multi-queue environment is to use *redundant requests*, where the users submit several requests simultaneously to multiple batch schedulers on behalf of a single job submission. Once, one of these requests is granted access to compute nodes, the others are canceled by the user. In the work by Casanova et. al. [12], the effect of redundant batch requests on different aspects has been discussed. While redundant requests have been shown to improve response times of jobs, it was also shown that batch system middleware may not be able to handle high loads due to the redundant requests, and may be complex to implement for practical purposes. We use redundant requests strategy as a baseline for evaluating our methods. In the work by Subramani et. al. [13], scheduling by the meta-scheduler is done based on the current load in the system for homogeneous systems. They define load as the ratio of (sum of the cpu times of the queued jobs + sum of the remaining cpu times of the running jobs) and the total system size. They first propose a "least loaded" scheme in which a greedy strategy is followed and the job is submitted to a particular queue/site with the least load. They also propose a "k-distributed" scheme in which the job is submitted to k least loaded queues/sites. This is similar to the redundant batch requests, except that, the job is submitted to some subset of the total number of queues/sites, based on the load on the system. While considering the load on the systems, their work does not consider the characteristics of the job to be scheduled. In our method, we explicitly consider the predictions of the queue waiting times for the scheduling of the job to the appropriate queue/site. Hence, the fitness of the current job to the queue/site is evaluated. The efforts by Sabin et al. [14] and Li et al. [15] discuss meta-scheduling in heterogeneous grid environments assuming knowledge of execution times on different systems. Our work focuses on homogeneous systems in which all processors associated with all the queues have the same speed, and hence response times of jobs are minimized by minimizing only the queue waiting times.

3 Predictions of Queue Waiting Times

3.1 Identifying Quick Starters

In our previous work [4], we have developed strategies for predictions of quick starters. For a given target job, our method splits the history for a target job into near, mid and long term history based on similarity of processor occupancy states. A processor occupancy state at a given instance denotes the allocation of the processors to the jobs executing at that instance. The method then finds similar jobs in the near, mid and long term history in terms of request size and estimated run time. The basis of identifying quick starters using near-term history is that by looking at jobs with similar characteristics in the near-term history with similar processor occupancy states and checking if those jobs have potentially been backfilled, it can be predicted if the target job can be backfilled and hence marked as a quick starter. For mid-term history, our method also considers the availability of free nodes for accommodating the request of the

target job, and position of the job in the queue in terms of request size and estimated runtime.

3.2 Predicting Queue Waiting Time Ranges

In this work, we propose a method which predicts either continuous or disjoint ranges of wait times for a job, with each range associated with the probability of the actual wait time lying in that range. The wait time prediction problem can be formulated as a supervised machine learning problem, which uses the history jobs, their feature vectors and their wait times as training set for future wait time predictions. Any job submitted to the system is defined by a set of features associated with the job. This set of features forms the *feature vectors*. The features associated with the job can be broadly classified into three categories. 1. Job Characteristics: These are the core characteristics associated with the job and are provided by the user at the time of target job submission. These include request sizes and estimated run times for the jobs. 2. Queue States: These are the properties associated with the batch queues and the jobs currently waiting for execution at the time of the arrival of the target job. 3. Processor States: These are the properties associated with the processor occupancies by the jobs currently executing in the system at the time of the arrival of the target job. In our work, we define a total of 19 features for a job's feature vector. Table 1 shows the list of 19 features along with the categories of the features.

From the table, we can see that we consider an extensive set of features to represent a job. We consider features related to the system states in addition to the job characteristics. We also consider features that rank the target job in relation to the jobs in the queue and the jobs in execution. These include:

- $queue_jobrank_{req_size}$,
- $queue_jobrank_{ert}$,
- $queue_jobrank_{cputime}$,
- $proc_{remain_cputime_lower_req_size}$,
- $proc_{remain_cputime_lower_ert}$, and
- $proc_{remain_cputime_lower_cputime}$.

While considering a small feature space would reduce the time taken for training and predictions, we found that the features we consider are essential in describing the job and system properties and to adequately capture similarities between two jobs. We arrived at this feature list by starting with only the features related to request sizes and estimated run times, and found that the resulting set leads to false similarities between two jobs. Hence, we included other features including job ranks and demand cpu times. While considering features to be included in the feature vector, we exclude features which are derivable from or dual of the already existing features in the feature set. For example, free nodes is derivable from the feature, occupied nodes, while the sum of the elapsed times of running jobs in the system is a dual feature of the sum of the remaining times of running jobs.

Table 1. Job Features

Feature	**Type**	**Description**
request_size	Job	No of processors requested by the user for the target job
ert (estimated run time)	Job	The approximate estimation of runtime provided by the user for the target job
$queue_jobrank_{req_size}$	Queue	The position of the target job in the list of waiting jobs in the queue at the time of its entry sorted in increasing order of request sizes
$queue_jobrank_{ert}$	Queue	The position of the target job in the list of waiting jobs in the queue at the time of its entry sorted in increasing order of ert's
$queue_jobrank_{cputime}$	Queue	The position of the target job in the list of waiting jobs in the queue at the time of its entry sorted in increasing order of cpu times (ert*$requestsize$)
$queue_{demand_cputime}$	Queue	The sum of the demand cpu times (ert*$requestsize$) of the waiting jobs in the queue
n_{queue}	Queue	The number of jobs waiting in the queue
$queue_{demand_cputime_lower_req_size}$	Queue	The sum of the demand cpu times (ert*$requestsize$) of the waiting jobs in the queue with lower *requestsize* than the target job
$queue_{demand_cputime_lower_ert}$	Queue	The sum of the demand cpu times (ert*$requestsize$) of the waiting jobs in the queue with lower *ert* than the target job
$queue_{demand_cputime_lower_cputime}$	Queue	The sum of the demand cpu times (ert*$requestsize$) of the waiting jobs in the queue with lower cpu times (ert*$requestsize$) than the target job
$proc_jobrank_{req_size}$	Processor	The position of the target job in the list of running jobs in the system at the time of its entry sorted in increasing order of request sizes
$proc_jobrank_{ert}$	Processor	The position of the target job in the list of running jobs in the system at the time of its entry sorted in increasing order of ert's
$proc_jobrank_{cputime}$	Processor	The position of the target job in the list of running jobs in the system at the time of its entry sorted in increasing order of cpu times (ert*$requestsize$)
$proc_{remain_cputime}$	Processor	The sum of the remaining cpu times (ert*$requestsize$ - $elapsed_{time}$*$requestsize$) of the running jobs in the system
n_{proc}	Processor	The number of jobs running in the system
$proc_{remain_cputime_lower_req_size}$	Processor	The sum of the remaining cpu times (ert*$requestsize$ - $elapsed_{time}$*$requestsize$) of the running jobs in the system with lower *requestsize* than the target job
$proc_{remain_cputime_lower_ert}$	Processor	The sum of the remaining cpu times (ert*$requestsize$ - $elapsed_{time}$*$requestsize$) of the running jobs in the system with lower *ert* than the target job
$proc_{remain_cputime_lower_cputime}$	Processor	The sum of the remaining cpu times (ert*$requestsize$ - $elapsed_{time}$*$requestsize$) of the running jobs in the system with lower cpu times (ert*$requestsize$) than the target job
occupied_nodes	Processor	The total number of nodes in the system that has been occupied by the running jobs

Defining Job Similarity In order to define similarity between a target job and the history job, we use their respective feature vectors and a distance metric called *Heterogeneous Euclid Overlap Metric (HEOM)* [16] for computing the distance between the target and the history job. Given two feature vectors, X and Y, each of size n, the HEOM distance between the two vectors is computed as $HEOM_{distance} = \frac{\sum_{i=1}^{n} ||X_i - Y_i||}{\sum_{i=1}^{n} (X_i + Y_i)}$. Instead of normalizing every element of the feature vector by its maximum value, the $HEOM_{distance}$ uses the original feature vectors and obtains a distance between 0 and 1. Hence, the $HEOM_{distance}$ metric can be efficiently used to find the difference between two feature vectors where each element in a vector can have different ranges.

Dynamic k-Nearest Neighbors The k-Nearest Neighbors is a popular learning algorithm in which the nearest k history jobs, in terms of their distance from the target job, are chosen as similar jobs. The wait times of the similar jobs are used for predictions for the target job. An important aspect of this learning algorithm is choosing of the value of k. In our method, we choose the value of k dynamically for every target job.

For a given target job, we compute the $HEOM_{distance}$ between the target job and each job in the history. We define the value of k as the number of history jobs within 5% of the distance from the target job. Thus, k is equal to the number of jobs in the history with a $HEOM_{distance}$ of less than or equal to the value of 0.05. If we do not find any jobs in the history within 0.05, we look for job within 0.1, 0.15 and so on until we find at least one similar job within any one of these thresholds. The higher the threshold value, the lower will be the similarity between the target and the history jobs.

Wait times of these k nearest neighbors are used to obtain a point wait time prediction for the target job. We explored three primary techniques for computing the point wait time of the target job.

1. Avg_{Wt}: Weighted Average using the inverse of the HEOM distance as a function of weight. This is computed using Equation 1.

$$Avg_{Wt} = \frac{\sum_{i=1}^{k} Weight_i * Wait_i}{\sum_{i=1}^{k} Weight_i} \tag{1}$$

   where, $Weight_i$ = inverse of HEOM distance between the target job and the ith history job, and $Wait_i$ = actual wait time of the ith history job.
2. Reg_{lin} : Linear Regression of the feature vectors and the wait times.

3. Reg_{non_lin} : Non Linear Regression of the feature vectors and the wait times using a non-linear kernel function of degree equal to the number of features (which is 19 in our case). We have explored three different kernel functions in our work, namely, polynomial, sigmoid and the radial basis functions.

For performing the linear and non-linear regressions we use the python libraries provided for regression methods in [17]. The weighted average and regression techniques assign weights, explicitly and implicitly, respectively, for each job in the history depending on the similarity of the job to the target job.

Multi-class Probabilities The point predictions obtained using kNN method can generally be highly inaccurate for queue waiting times due to several factors including small number of similar jobs in the history, large distance between the target job and the most similar job or the fact that even the most similar jobs in the history can have high variations in wait times. Hence we attempt to provide predictions of ranges of queue waiting times. For providing a range of wait times, we divide the wait times of a job into multiple classes. In our work, we use the classes shown in Table 2 since these classes are commonly observed in many supercomputing traces [18]. For a given target job, we first compute the predicted class based on the predicted point wait time obtained from using k-NN. For example, if the predicted point wait time is 1 hour and 15 minutes, the predicted class is class 2 since its range is 1 to 3 hours.

Table 2. Class Ranges

Class	Wait Time Ranges
1	less than or equal to 1 hour
2	1 hour to 3 hours
3	3 hours to 6 hours
4	6 hours to 12 hours
5	12 hours to 24 hours
6	greater than 24 hours (or 1 day)

As a next step, we use another machine learning technique, *Support Vector Machine (SVM)*, for multi-class classification of a target job. SVM is one of the most popular classification based learning algorithms. SVMs can efficiently perform non-linear classification using the "kernel trick", implicitly mapping their inputs into high-dimensional feature spaces. SVMs are also computationally less expensive, and handles over-fitting better than the other methods. It is a supervised learning algorithm which takes as input the training set consisting of history jobs, their feature vectors and the classes of the jobs based on their actual waiting times. It also takes as input the target jobs and their feature vectors. For a target job, based on the training model, it outputs the probabilities of the target job's waiting time belonging to the different classes. In order to get

the class probabilities, we feed the entire set of jobs in the history to the SVM Multi-Class Classification library [17] for the training set. Since SVM training can be time consuming (order of few tens of seconds for each target job), we use SVM training only at regular intervals and not for every target job. For our experiments, SVM training was done after every 5000 jobs.

Our next goal is to convert the point wait time obtained from the kNN method into more reliable ranges. This is done in two steps. In the first step, we consider the predicted class, X, corresponding to the point wait time predicted by kNN. This class gives a single range of wait times. As reported in our experiments in Section 5, this single class prediction results in prediction errors in large number of cases. Hence we attempt to provide two class ranges by considering the two immediate neighboring classes of X. We obtain the probabilities of these three classes using the SVM classifier. Two of these three classes with the two highest probabilities are given as the ranges of wait times for the target job, along with their normalized probabilities. The normalized probability of a class is obtained by the ratio of its probability given by SVM and the sum of the probabilities of the two classes. It has to be noted that our method only gives the ranges of wait times as the output for the user. The probabilities are used internally by the meta-scheduler described in the next section. If the class X is either the first or last class (i.e. less than or equal to one hour or greater than 1 day), then our method provides only two class ranges.

Two primary reasons guide our methodology of using the predicted class obtained by the dynamic k-NN method and its neighboring classes for the selection of the output predicted classes (or ranges). Using the SVM probabilities directly can at times lead to a situation in which the top two or three classes with highest probabilities may be separated by more than one class, i.e., widely separated, and hence may not be intuitive to the user. Hence, we obtain a predicted class using the dynamic k-NN and only try to make some minor corrections in the predictions using the SVM class probabilities. Another reason is that in some cases the SVM class probabilities may not give a unanimous choice of a class as a clear leader. In such scenarios, choosing two or three classes with highest probabilities may not prove to be effective as the absolute probabilities for each of them may be quite low. Hence, we reduce the candidate classes for prediction to three using k-NN, and use only the probabilities among the three.

The entire algorithm followed in our *Multi-Class Wait time ranges* method is illustrated in the flowchart shown in Figure 1.

4 Metascheduling

A single supercomputing system typically consists of multiple batch queues that differ in terms of the processor and execution time requirements of the jobs to cater to the needs of different jobs. A user chooses a batch queue among the many available batch queues when submitting a high performance computing job for problem solving. This choice is typically made by the user based on his limited experience with the application on some of the systems available.

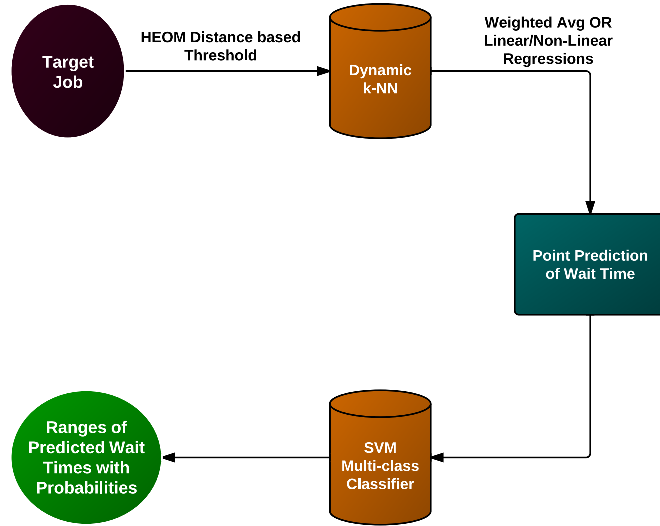

Fig. 1. Multi-Class Methodology for providing predicted wait time ranges with probability

The user mostly chooses the system on which he has had the experience of best performance or minimum queue waiting time. Most of the supercomputing systems have static scheduling policy of distributing the jobs among the queues. This is primarily done based on the request size and the estimated run time of the job provided by the user.

Meta-scheduler is a top-level scheduler that distributes jobs among multiple systems to optimize costs including execution time or system utilization. We have developed a meta-scheduler that uses the quick starter identification and stochastic predictions of the queue wait times for selecting an appropriate queue/site among a given set of queues for the job. The primary objective of our meta-scheduler is to reduce the average wait times of the jobs submitted to the queues. The system model considered in this work is that all the processors corresponding to the queues are homogeneous, i.e., the execution times of a job when submitted to different queues are the same.

We have developed an algorithm called *Least_Predictedqw* which, for a given target job, identifies the queue/site with the least predicted queue waiting time and schedules the job to that particular queue/site for the job execution. Let us consider, for a given target job, n different queues/sites for scheduling the job. As a first step, we use our first prediction method, PQStar, to find the number of queues, m, $(m < n)$, for which the target job is predicted as a quick starter. A three-way decision is then made:

1. if $m = 1$, the particular queue/site in which the target job is predicted as a quick starter is chosen for scheduling the job for its execution.

2. if $m = 0$ or $m = n$, then the n queues are added to a candidate queues/site list, since any of the n queues can be chosen for job scheduling.
3. if $1<m<n$, then the target job is predicted as a quick starter in the subset (of size m) of total queues/sites in the system. In this case, then the m queues are added to the candidate queues/site list.

As a next step, we consider the queues/sites in the *candidate queues/sites list* and obtain the queue waiting time ranges and their corresponding probabilities by using our machine learning based methods described in Section 3. Then, for each of the queue/site on the candidate list, we compute their corresponding expected wait times as $Exp_Val = \sum_{i=1}^{n} Prob_i * Mean_Wait_i$, where, $Prob_i$ = probability of the ith range & $Mean_Wait_i$ = mean waiting time of all the similar jobs in the history in the ith range. We then choose the queue that has the least expected queue waiting time.

5 Experiments and Results

5.1 Evaluation of Predictions

In this section, we show the evaluation results of our prediction methodology and its effectiveness in successfully providing ranges of queue waiting times as predictions.

Experimental Setup For evaluations of our predictions, we have developed a discreet event simulator. The simulator creates a simulated environment of the jobs waiting in the queue and running on the system at different points of time. It keeps track of the jobs submitted to the system, and maintains their attributes including arrival times, wait times, actual runtimes and request sizes. It does not simulate the actual scheduling algorithm used, thus avoiding assumptions about the underlying scheduling algorithm. The user can invoke the simulator with a supercomputing job trace/log in the Standard Workload Format (SWF) [19] as input, and obtain predicted queue waiting time of a new job. The simulator creates the simulated environment of jobs in the system using the statistics available in the log. The simulator is triggered by three primary events corresponding to job arrival, job beginning to execute and job termination. Whenever a job arrives, it is added to a waiting queue maintained by the simulator. As soon as a job's wait time is over and it starts executing, it is removed from the waiting queue and added to a running list in the simulator. Also at this time, the free nodes available in the system is decremented by the value equal to the job's request size. Once a job which is running completes its execution, it is removed from the running list and the free nodes available in the system is incremented by the value equal to the job's request size. This process is repeated for each job and thus a simulated system state is created using which we extract

the processor state and the queue properties that are needed for our prediction model.

We performed experiments for six supercomputing traces obtained from the Parallel Workload Archive [18]. For each supercomputing trace in our experiments, we performed predictions for all the jobs starting from the 10001th job up to a maximum of 50000 jobs or the end of the log. Each of the jobs in this set constitutes the evaluation data for which predictions were made. For a given target job for which waiting time is predicted, all the jobs submitted prior to it constitute the history. For our experiments, we limited the history size to 5000 jobs for maintaining the time taken for a prediction to within few microseconds. Once the target jobs start their execution and their wait times are known, they are added to the set of history jobs.

Point Predictions with k-NN We first show the prediction accuracies for point predictions of queue wait times with different techniques using similar jobs obtained from k-NN. We compute the percentage difference in predicted and actual response times for each job, where response time is the sum of queue waiting time and execution time. For the execution time, we consider the estimated run time (ERT) supplied by the user to be equal to the actual execution time. Hence the percentage predicted error in response time is calculated as $PPE_{rt} = \frac{|predictedwaitingtime - actualwaitingtime|}{actualresponsetime}$. This metric determines the amount of impact of the prediction errors on jobs of different lengths or execution times. Table 3 shows the average percentage prediction error in response time for different supercomputing traces [18] with the five techniques, namely, weighted average, linear regression, and non-linear regression with polynomial, radial basis functions (RBF) and sigmoid functions. As can be seen from the results, evaluations with the different techniques show very little variations in predictions. Thus our overall method of obtaining point predictions using k-NN gives similar results irrespective of the technique used to obtain the predictions. Based on these results, we use *weighted average* in our further experiments due to its relatively lesser computational complexity compared to the linear/non-linear regressions. The high percentage prediction errors shown in the table also indicate the challenges in predictions in batch systems due to non-deterministic job arrivals and terminations, and hence the less usefulness of point predictions when compared to range predictions for queue waiting times.

Single Class Accuracy We first show the results of the single class accuracy predictions for our method and the IBL method. Success is defined as the percentage of jobs for which the predicted class (obtained from the dynamic k-NN method or the point wait time prediction provided by IBL) is the true class of the job. Table 4 shows the percentage of jobs with successful prediction of the true class with a single class accuracy, for all jobs and for non-quick starters. We find that both our method using weighted average and IBL technique consistently give better predictions than QBETS. This is because these methods consider both the queue and processor states in addition to the job characteristics while

Table 3. Percentage Prediction Error in Response Time

Logs	$\mathbf{Avg_{wt}}$	$\mathbf{Reg_{lin}}$	$\mathbf{Reg_{poly}}$	$\mathbf{Reg_{rbf}}$	$\mathbf{Reg_{sig}}$
CTC	54	51	52	53	51
ANL	50	49	51	50	51
LANL	71	71	68	69	70
HPC2N	65	62	61	61	61
SDSC Blue	72	73	72	71	73
SDSC SP2	111	108	108	108	108

QBETS using its trace-based predictions considers only the job characteristics. We can also see that our methods can correctly give the single class predictions for up to 16% more number of jobs than IBL for all the jobs (Table 4(b)). But the performance improvement for the non-quick starters when compared to IBL is relatively lesser and our method can give correct predictions only for up to 4% more number of jobs for the non quick starters (Table 4(a)). This shows that with single class predictions, the accuracy that can be obtained can be low. Hence, it will be more useful to give the user with multiple classes (or ranges) of queue waiting times as predictions.

Table 4. Percentage of Jobs predicted with Single class accuracy

(a) Non Quick Starters

Logs	**QBETS**	**IBL**	$\mathbf{Avg_{wt}}$
CTC	19	28	30
ANL	19	29	30
LANL	10	26	28
HPC2N	13	22	23
SDSC Blue	15	24	28
SDSC SP2	30	33	35

(b) Overall

Logs	**QBETS**	**IBL**	$\mathbf{Avg_{wt}}$
CTC	4	47	59
ANL	18	59	73
LANL	58	61	76
HPC2N	16	48	64
SDSC Blue	37	57	74
SDSC SP2	8	48	64

Multi Class Accuracy We next show the results for predictions of two classes, that are determined using the methodology described in Section 3, for our

method and compare with the QBETS and IBL methods. Since IBL method gives only a point wait time as prediction, for the purpose of multi class accuracy comparisons, we used an extended version of IBL. In this version, the point wait time predictions of IBL are used with our SVM multi-class classifier to obtain ranges of wait time predictions. This extension of point wait-time predictions to multi-class predictions is reasonable since the multiple classes are obtained in the neighborhood of the class to which the predicted point wait time belongs, both in our method using k-NN and the IBL method. Success is defined as percentage of jobs for which at least one of the two predicted classes (class with top two probabilities) is the true class of the job.

Figure 2 shows the percentage of jobs with successful prediction of the true class with a two class accuracy, for all jobs and for non-quick starters. We can see that our methods can correctly give two class predictions for up to 17% more number of jobs overall for all the jobs, and up to 12% more number of jobs for the non quick starters, when compared to IBL. Also, overall for all the jobs, our method can correctly predict for about 77-87% of the jobs with a two class accuracy.

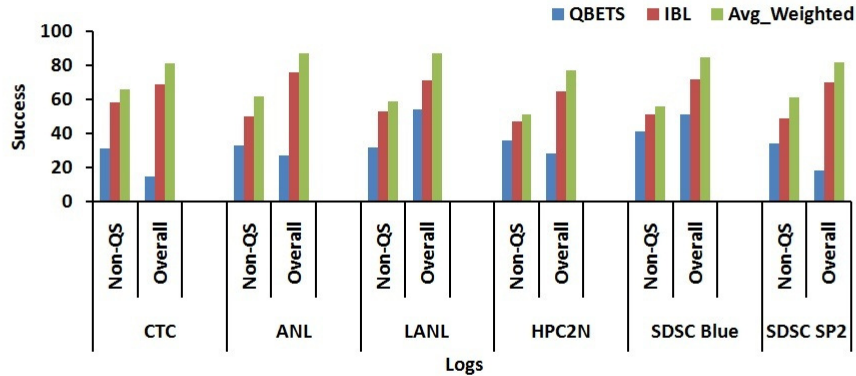

Fig. 2. Percentage of Jobs predicted with a Two class accuracy

Finally we also show the results for three class accuracy for all jobs and for non-quick starters. Success is defined as percentage of jobs for which at least one of the three predicted classes (predicted class obtained from the dynamic k-NN method and its neighboring classes) is the true class of the job. We found that our methods can correctly give three class predictions for up to 11% more number of jobs overall for all the jobs, and up to 8% more number of jobs for the non quick starters, when compared to IBL. Also, overall for all the jobs, our method can correctly predict for about 82-91% of the jobs with a three class accuracy. This confirms that even though our predictions for single class/range of wait times have low accuracy and are comparable to IBL, we achieve better success rate for predictions of multiple classes. This is primarily because of the

fact that, for a subset of the jobs where both our method and IBL method fail to provide good single class accuracy, our method gives the class that is closer (neighbor) to the true class to which the job belongs as the predicted class, when compared to the IBL methodology. This improvement is primarily due to the use of better similar jobs in history in our method obtained by the use of dynamic k-NN and the extended feature set. The multi-class predictions with probabilities are primarily intended for our metascheduler described in the next section. However, these class predictions by themselves can be useful for a user to obtain broad-level estimates and help in broad-level decisions for job submissions.

5.2 Evaluation of Metascheduling

In this section, we show the evaluation results of the meta-scheduling strategies and its effectiveness in reducing the queue waiting times incurred by the jobs.

Experimental Setup For experiments related to meta-scheduler, we use *GridSim* [20] for simulating the scheduling algorithm using the workload logs. *GridSim* allows modeling and simulation of entities in parallel and distributed computing systems-users, applications, resources, and schedulers for design and evaluation of scheduling algorithms. It allows the management and also provides a way to analyze algorithms on large-scale distributed systems. In order to simulate the various workloads on different queues/sites, we employ EASY-back filling [7] scheduling algorithm using the scheduling simulation provided by the *GridSim*. We provide the supercomputer logs in the SWF [19] format to the *GridSim* and also specify the scheduling algorithm to be simulated. Given these inputs, the *GridSim* simulates the scheduling algorithm for the given logs. Using the simulation results, we can obtain the actual wait times of the different jobs in the logs.

In order to simulate the meta-scheduling scenarios, we divide the system into different partitions, with each partition associated with certain maximum number of processors. After the meta-scheduler chooses a queue for a job execution, the job is submitted to the particular queue for its execution in the GridSim simulation, and the actual wait time incurred by the job is obtained. For performing the redundant batch request experiments, which we use as a comparison method, we use the simulator [21] built by Casanova for his work on redundant requests [12]. As an input we provide the supercomputer logs in the SWF [19] format. It is required to set the configuration file for the simulations such that all the jobs submitted to the queues resort to redundant requests and the requests are submitted to all the queues available in the system. We show results for both the real workload traces obtained from the Parallel Workload Archive [18] and certain synthetic traces which we have generated using the Parallel Workload Models [22]. We perform two kinds of experiments: *super experiments* and experiments with *synthetic traces*.

In the *super experiments*, we create a grid like environment in which we consider six different sites which are geographically distributed. For the six sites,

we used the real workload traces obtained from the Parallel Workload Archive [18] for logs of the six sites, namely, CTC, ANL, SDSC Blue, SDSC SP2, LANL and HPC2N. We simulate a meta-scheduler capable of submitting the jobs to any of these sites. Note that while the job arrivals and job parameters in the super experiments correspond to the actual traces, we submit the job traces of the different logs to GridSim's EASY-backfilling scheduler to obtain queue waiting times. This is due to the difficulty in simulating the exact scheduling algorithms with fine-level policies in GridSim including priorities to short jobs, priorities to certain queues on specific days, effect of draining the jobs due to reservations, system failures etc. Moreover, the fine-level details are not available for all the systems in the workload archive.

For experiments with *synthetic traces*, we generated 32 synthetic scenarios with synthetic job traces. We consider four queues in all the scenarios. The different scenarios are as follows. We used a total of four system sizes, namely, 1K, 16K, 128K and 256K. For each of the system sizes, we used two processor partitioning schemes, namely equal and unequal partitioning schemes The equal partitioning scheme has four processor partitions of size $X/4$ each, where X is the total number of processors. The unequal partitioning scheme also has four processor partitions of sizes $X/8$, $X/8$, $X/4$ and $X/2$. Finally, for each of the four system sizes and each of the two processor partitioning schemes, we simulate four different kinds of loads on the system. These loads are modeled by different inter-arrival times, based on the real inter-arrival times of real workloads obtained from Feitelson's workload archive [18]. With this, we generate synthetic workloads which have almost similar job arrival patterns as the real workloads. Table 5 shows the various average inter-arrival times for each of the queues that we have used. The configurations of the four queues referred in Table 5 are shown in Table 6.

Table 5. Inter-arrival times for jobs in the queues

Load	**Queue 1**	**Queue 2**	**Queue 3**	**Queue 4**
I	15 min	1 hr	2 hrs	2 hrs
II	15 min	1 hr	1 hr	1 hr
III	15 min	1 hr	2 hrs	5 hrs
IV	30 min	2 hrs	2 hrs	2 hrs

Hence we generate a total of $4(systemsizes) * 2(partitions) * 4(loads) = 32$ *scenarios*. These scenarios were carefully constructed after adequate survey of the various real systems in Feitelson's workload archive [18]. While a grid may not have a system ranging from 1K to 256K processors, these system sizes were chosen on the basis of the systems for which job traces were available in the workload archive. For each of the scenarios, we use the Parallel Workload Models [22, 23] for obtaining the synthetic traces. Using a workload model enables us to vary workload characteristics of the batch queues and study the variations of

response times. The workload model generates a job trace consisting of arrival times, processor requirements and user estimated execution times of the jobs. Jobs corresponding to each queue are generated in the system based on the processor request size and the estimated run time characteristics of the job, following the protocols of the queue configuration. Each queue has a specific configuration in terms of maximum request size and maximum runtime allowed for a job. The maximum request size in terms of the number of processors and the max running time of a job for each of the queue in a system of X processors is shown in Table 6.

Table 6. Queue Configurations

Queue ID	**Max Request Size**	**Max ERT**
I	X/32	12 hrs
II	X/16	18 hrs
III	X/8	1 day
IV	X/4	2 days

We compare our meta-scheduling strategy, *least_predictedqw*, with three strategies, namely, *static scheduling*, *least loaded*, and *redundant batch requests*. For our strategy *least_predictedqw* that uses predictions, we use our two-class predictions, explained in Section 3, since it gives better accuracy than single-class predictions and about the same accuracy as three-class predictions. We chose two-class predictions over three-class predictions since smaller number of classes imply tighter ranges of predictions. In our strategy, the metascheduler obtains the expected queue waiting time of a job for a given queue using multi-class predictions with probabilities. The estimated wait time is obtained as the weighted sum of the averages of the queue waiting times of the similar jobs in the classes, weighted by the probabilities for the classes. Our *least_predictedqw* then chooses the queue with the least expected waiting time for job submission. In the *static scheduling* strategy, the job will be scheduled for execution at the particular queue/site, where the job was originally submitted. In the *least loaded* technique proposed by Subramani et. al [13], scheduling by the meta-scheduler is done based on the current load in the system. They define load as the ratio of (sum of the cpu times of the queued jobs + sum of the remaining cpu times of the running jobs) to the total system size. A greedy strategy is then followed in which the job is submitted to the particular queue/site with the least load using the above load definition. In *redundant batch requests*, the job is submitted to all the queues and once the job starts its execution in any one of the queues, then the rest of the submissions are canceled. While a middleware may not be able to handle high loads due to redundant submissions, we use this technique as a baseline for evaluating the goodness of our method.

Results with Super Experiments We first show the effectiveness of our least_predictedqw method over other methods in terms of the average wait time incurred by all the jobs submitted to the system. Figure 3 shows the average wait times of jobs for the different meta-scheduling methods. The figure shows that our methods gives up to 54% and 33% reduction in the average wait time compared to the static and least loaded methods, respectively. It can also be seen that our method is closest to the baseline method of redundant batch requests.

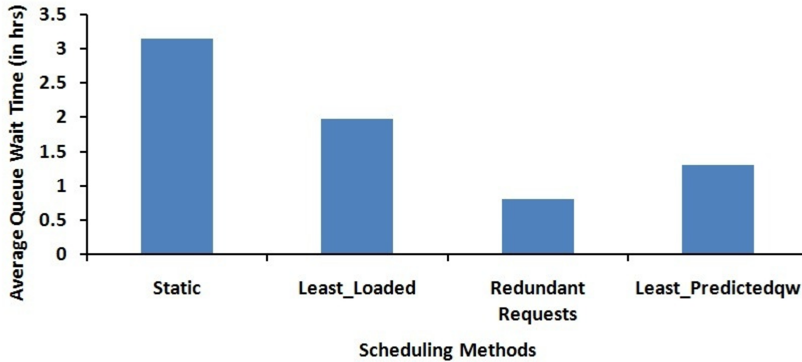

Fig. 3. Super Experiments Evaluation: Average Wait Time

We also show the distribution of jobs across the various sites. An efficient distribution should be able to distribute in such a way that the number of jobs scheduled to a site is directly proportional to the system size associated with the site. Table 7 shows the distribution of the jobs across the sites for the various methods. The table also gives information regarding the system size. We can see that the proportionality is more evident in case of our Least_Predictedqw method, when compared to the other methods. For example, CTC has comparatively lesser system size compared to LANL, but the number of jobs scheduled in CTC is more by the Least_Loaded method than in LANL, while in our method the LANL has more jobs scheduled than CTC.

Results with Synthetic Traces We now show some of the overall statistics over all the 32 synthetic scenarios. Table 8 shows the absolute statistics for each of the methods. In general, our method resulted in 35-94% decrease in average wait time when compared to the static method, and 14-78% decrease in average wait time when compared to *least_loaded* method. Our method also resulted in 8-80% increase in the number of jobs with wait times less than one hour, and 6-99% decrease in the number of jobs with wait times more than a day, when compared to the other methods.

Table 7. Super Experiments Evaluation: Job Distribution - percentage of the total jobs scheduled in each of site

Sites	System Size	Static (%)	Least_Loaded (%)	Least_Predictedqw (%)
CTC	430	22.9	13.7	10.4
ANL	163840	20.7	53.4	52.1
LANL	1024	21.7	11.2	18.3
HPC2N	240	5.7	7.6	2.0
SDSC Blue	1024	21.0	11.1	18.4
SDSC SP2	128	7.9	2.7	1.4

Table 8. Synthetic Experiments Evaluation: Overall Statistics (Absolute)

Attributes	Static	Least_Loaded	Least_Predictedqw
Avg wait time (in minutes)	196	105	52
% jobs with wait time $\leq$ 1 hr	68	76	89
% jobs with wait time $\geq$ 1 day	3.2	1.4	0.5

6 Conclusions and Future Work

We have developed a machine learning based hierarchical prediction strategy for prediction of ranges of queue waiting times and probabilities. We used these predictions and probabilities in a meta-scheduling strategy that distributes jobs to different queues/sites in a multi-queue/grid environment for minimizing wait times of the jobs. Our experiments with different production supercomputer job traces show that our prediction strategies can give correct predictions for about 77-87% of the jobs, and also result in about 12% improved accuracy when compared to the next best existing method. Our experiments with our meta-scheduling strategy using different production and synthetic job traces for various system sizes, partitioning schemes and different workloads, show that our meta-scheduling strategy gives much improved performance when compared to existing scheduling policies by reducing the overall average queue waiting times of the jobs by about 47%.

While we have used SVM for predictions, incremental machine learning approaches can also be explored since they help in reducing the times for training and predictions. While the focus of this work is predictions of queue waiting times and metascheduling to reduce average wait times, we plan to develop techniques for predictions of execution times in order to predict total response times. We also plan to extend the meta-scheduling strategies to heterogeneous systems. The primary challenge in metascheduling for heterogeneous systems is to predict the runtime of a user job on different systems with different configurations, in addition to predicting the queue waiting times. While prediction

of the run time for a job within a site can be made by considering similar job submissions by the same user in the history, the prediction can be extended to the other platforms by cross-platform performance modeling. We would also like to extend the meta-scheduling strategies that use stochastic predictions of both the wait times and the run times to select the appropriate number of resources for job executions in addition to the selection of the queue/site.

7 Acknowledgments

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

References

1. "IBM Load Leveler," http://www.redbooks.ibm.com/abstracts/sg246038.html.
2. "PBS Works," http://www.pbsworks.com.
3. "Tera Grid Karnak Prediction Service," http://karnak.teragrid.org/karnak/index.html.
4. R. Kumar and S. Vadhiyar, "Identifying Quick Starters: Towards an Integrated Framework for Efficient Predictions of Queue Waiting Times of Batch Parallel Jobs," in *In the Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, Shanghai, China, May 2012.
5. W. Smith, I. T. Foster, and V. E. Taylor, "Predicting Application Run Times Using Historical Information," in *IPPS/SPDP '98 Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, 1998, pp. 122–142.
6. W. Smith, V. E. Taylor, and I. T. Foster, "Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance," in *IPPS/SPDP '99/JSSPP '99: Proceedings of the Job Scheduling Strategies for Parallel Processing*, 1999, pp. 202–219.
7. D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, "Parallel Job Scheduling - A Status Report," in *JSSPP'07 Proceedings of the 13th international conference on Job scheduling strategies for parallel processing*, 2004, pp. 1–16.
8. D. Nurmi, J. Brevik, and R. Wolski, "QBETS: Queue Bounds Estimation from Time Series," in *JSSPP'07 Proceedings of the 13th international conference on Job scheduling strategies for parallel processing*, 2007, pp. 76–101.
9. J. Brevik, D. Nurmi, and R. Wolski, "Predicting Bounds on Queuing Delay for Batch-Scheduled Parallel Machines," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006, pp. 110–118.
10. H. Li, D. L. Groep, and L. Wolters, "Efficient Response Time Predictions by Exploiting Application and Resource State Similarities," in *GRID '05 Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, 2005, pp. 234–241.
11. H. Li, J. Chen, Y. Tao, D. L. Groep, and L. Wolters, "Improving a Local Learning Technique for Queue Wait Time Predictions," in *CCGRID '06 Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, 2006, pp. 335–342.

12. H. Casanova, “Benefits and Drawbacks of Redundant Batch Requests,” *Journal of Grid Computing*, vol. 5, 2007.
13. V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan, “Distributed Job scheduling on Computational Grids Using Multiple Simultaneous Requests,” in *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, 2002, pp. 359–366.
14. G. Sabin and M. Lang, “Moldable Parallel Job Scheduling using Job Efficiency: An Iterative Approach,” in *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), in conjunction with ACM SIGMETRICS*, 2006.
15. H. Li, D. Groep, and L. Wolters, “Mining Performance Data for Metascheduling Decision Support in the Grid,” *Journal for Future Generation Computer Systems - Special section: Data mining in grid computing environments*, vol. 23, no. 1, pp. 92–99, 2007.
16. D. Wilson and T. Martinez, “Improved Heterogeneous Distance Functions,” *Journal of Artificial Intelligence Research*, vol. 6, p. 134, 1997.
17. “Machine Learning in Python,” http://scikit-learn.org/stable.
18. “Parallel Workload Archive,” http://www.cs.huji.ac.il/labs/parallel/workload/logs.htm.
19. “Standard Workload Format,” http://www.cs.huji.ac.il/labs/parallel/workload/swf.htm.
20. R. Buyya and M. Murshed, “Gridsim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing,” *CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE (CCPE*, vol. 14, no. 13, pp. 1175–1220, 2002.
21. “Redundant Batch Requests Simulator,” http://sourceforge.net/projects/redsim.
22. “Parallel Workload Models,” http://www.cs.huji.ac.il/labs/parallel/workload/models.htm.
23. U. Lublin and D. G. Feitelson, “The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs,” *Journal of Parallel and Distributed Computing*, vol. 63, no. 11, pp. 1105–1122, 2003.