

Mining Performance Data for Metascheduling Decision Support in the Grid

Hui Li^a *, David Groep^b, and Lex Wolters^a

^aLeiden Institute of Advanced Computer Science (LIACS), Leiden University, PO Box 9512, 2333 CA, Leiden, The Netherlands

^bNational Institute for Nuclear and High Energy Physics (NIKHEF), PO Box 41882, 1009 DB, Amsterdam, The Netherlands

Abstract: Metaschedulers in the Grid need dynamic information to support their scheduling decisions. Job response time on computing resources, for instance, is such a performance metric. In this paper, we propose an Instance Based Learning technique to predict response times by mining historical performance data. The novelty of our approach is to introduce policy attributes in representing and comparing resource states, which are defined as the pools of running and queued jobs on the resources at the time of making predictions. The policy attributes reflect the local scheduling policies and they can be automatically discovered using genetic search. An extensive empirical evaluation is conducted to validate our technique using real workload traces, which are collected from the NIKHEF production cluster on the LHC Computing Grid and Blue Horizon in the San Diego Supercomputer Center (SDSC). The experimental results show that acceptable prediction accuracy can be achieved, where the normalized average prediction errors for response times are ranging from 0.57 to 0.79.

Keywords: Response Time Predictions, Instance Based Learning, Metascheduling, Grid

1. Introduction

Large scale Grids typically consist of many heterogeneous and geographically distributed resources. As an example, the LHC Computing Grid (LCG) currently has approximately 140 sites in 34 countries with a total number of 12,516 CPUs and 5 petabytes storage. Metascheduling in such an environment raises a serious challenge and many scheduling algorithms, architectures and systems have been proposed [1,3,16]. Scheduling at the meta level differs from local scheduling in that metaschedulers do not have control over the resources. Instead, metaschedulers make decisions on behalf of users and hand jobs over to the local resource management systems. There is one common aspect in this process despite the diversity of Grid scheduling instances, namely, the goodness of scheduling decisions depends heavily on the quality of information available about the resources. There is relatively static information such as machine types, number of CPUs and storage capacity. This

can be obtained by monitoring tools via Grid information services. There is also more dynamic information such as job response times. This type of information is very important to support the metascheduling decisions, but is not available by only monitoring. It must be predicted based on historical data recorded on the resources.

The main theme of this paper is about job response time predictions on computing resources, and our approach is based on mining the historical performance data. We believe that knowledge about local scheduling policies can be discovered in the data and this knowledge can be utilized in predictions. Techniques from statistical data mining can help us getting there. Specifically, we investigate how an Instance Based Learning technique is applied in predictions, how a genetic algorithm is used for parameter optimization, and elaborate our design choices. We focus on resources such as space-shared parallel supercomputers, clusters, and study workload traces recorded on them.

The rest of the paper is organized as follows: Section 2 introduces job response time predictions and

*Corresponding author (hui.li@computer.org).

discusses the related work in this area. Section 3 defines job similarity and resource state similarity, which are the two key concepts in our technique. Section 4 elaborates the IBL-based prediction algorithm, including the distance function and the induction models. Section 5 describes the design and construction of the genetic algorithm for parameter optimization. Section 6 presents the empirical evaluation and analysis using real workload traces. Conclusions and future work are discussed in Section 7.

2. Response Time Predictions

Response time of a job is defined as the time elapsed from its submission till completion on a resource. Two metrics need to be estimated for the response time: one is how long a job executes on the resource (application run time), the other is how long the job waits in the queue before starting (queue wait time). A popular approach is to derive predictions from similar observations in the past, and the historical data available in site workload traces naturally serves as the basis for such study.

Techniques have been proposed for predicting application run times using historical information. In [13] “templates of attributes” are defined for categorizing historical jobs and statistical techniques like mean or linear regression are applied to generate predictions. In [6] Instance Based Learning (IBL) techniques are investigated for run time predictions. IBL uses historical data “near” the query point to build a local model for approximation. A proper distance metric has to be defined to measure the distances between data instances. In fact the IBL algorithm is a generalization of the template approach, in which distances are simplified to binary values (belong or not belong to a specific category).

For queue wait times, the basic idea of most techniques is based on scheduler simulation. In [15] scheduling algorithms like FCFS, LWF, and backfilling are simulated for predicting queue wait times, where application run times are estimated using the template approach. In [7] simulation is also used to predict queue wait times for a policy-based scheduler called Maui. Although relatively better prediction accuracy can be achieved, several major drawbacks remain for the simulation approach. Firstly, it cannot meet the real-time requirements in the Grid

brokering process. Secondly, it is not scalable since there are different types of local scheduling systems deployed on different Grid sites. Some sites have schedulers with combinations of basic scheduling algorithms, and most sites enforce different kinds of policies in their own fashion.

As an alternative, we are trying to derive queue wait times also from historical observations. Our assumption is that “similar” jobs under “similar” resource states would most likely have similar waiting times, given that the scheduling algorithm and policies remain unchanged for a reasonable amount of time. Similar ideas has been studied in [14], in which summary statistics about the resource state (e.g. free CPUs, number of running jobs) is used as attributes for defining templates. The template approach to estimate application run times as described above can be applied similarly for waiting times. Our work distinguishes from it in two aspects: Firstly, we introduce attributes that reflect scheduling policies to represent resource states in a more fine-grained level for similarity comparison, and these policies can be automatically discovered via genetic search. Secondly, we use Instance Based Learning as the common framework for both application run times and queue wait times. We elaborate our approach in the following sections.

3. Similarity Definition

The key problem is how to define similarity to compare jobs. Table 1 shows the representative job attributes recorded in workload traces. For job run times, some of these attributes can be naturally used for similarity definition. For queue wait times, however, new attributes need to be defined as the waiting time of a job is typically resulted by interactions among the job, other jobs on the resource, and the local scheduler. We introduce the definitions for job similarity and resource state similarity as follows.

3.1. Job Similarity

Seven recorded attributes are considered to define job similarity. They are “group name” (g), “user name” (u), “queue name” (q), “job name” (e), “number of CPUs” (n), “requested run times” (r), and “arrival time of day” (tod). Depending on the availability, Any potentially useful attribute such as node speed and executable arguments can be added to

Abbr.	Job Attribute	Type	Abbr.	Job Attribute	Type	Abbr.	Job Attribute	Type
g	group name	nominal	n	#CPUs	numeric	m	used memory	numeric
u	user name	nominal	r	req. run time	numeric	rm	req. memory	numeric
q	queue name	nominal	tod	arrival time	numeric	rt	run time	numeric
e	job name	nominal	s	exit status	numeric	qt	queue wait time	numeric

Table 1
Representative job attributes recorded in workload traces.

Abbreviation	State Attribute	Value Calculation
VRunJobs	Vector of categorized number of running jobs	$V_i = Nr_i$
VQueueJobs	Vector of categorized number of queuing jobs	$V_i = Nq_i$
VAlreadyRun	Vector of categorized sum of past run time multiplied by #CPUs of running jobs	$V_i = \sum_{j=1}^{Nr_i} alrunt(j) \times cpu(j)$
VRunRemain	Vector of categorized sum of estimated remaining run time multiplied by #CPUs of running jobs	$V_i = \sum_{j=1}^{Nr_i} remaint(j) \times cpu(j)$
VAlreadyQueue	Vector of categorized sum of past queue time multiplied by #CPUs of queued jobs	$V_i = \sum_{j=1}^{Nq_i} alrquet(j) \times cpu(j)$
VQueueDemand	Vector of categorized sum of estimated run time multiplied by #CPUs of queued jobs	$V_i = \sum_{j=1}^{Nq_i} demandt(j) \times cpu(j)$

Table 2
Defined resource state attributes. V_i : the value of the i th category, Nr : number of running jobs, Nq : number of queued jobs. The template for categorization is a subset of $\langle group, user, queue \rangle$.

this list. The pre-selected attributes are mostly self-explanatory by their names and they have two main types, namely, nominal (g, u, q, e) or numeric (n, r, tod). In the Instance Based Learning algorithm described later, similarity between jobs are formulated by a distance function composed of attributes. Jobs with smaller distances are considered more similar. For instance, jobs from the same group, say “bioinfo”, and with same executable name called “proteinmatch” will have smaller distances than those who have nothing in common, therefore have a higher chance being used for predictions of the same kind. To make a good distance function, we employ a genetic search to weight attributes according to the metric being predicted (application run time or queue wait time).

3.2. Resource State Similarity

We define a *resource state* as the pool of running and queued jobs on the resource at the time of making a prediction. More issues arise when measuring similarities of resource states and we need attributes to characterize them. Firstly, since a resource state con-

sists of a set of jobs, attributes have to be defined to represent the resource states in a way that they can be compared properly. Secondly, policy attributes that reflect the scheduling policies have to be embedded into the state attributes so that local scheduling information can be included.

Three attributes are defined as the candidate policy attributes, namely, *group name*, *user name*, and *queue name*. These job credential attributes are most frequently used in defining scheduling policies. For instance, group, which is also referred to “Virtual Organization” in the Grid, is a popular choice to make scheduling policies based on various QoS requirements. Administrators may define multiple queues and assign different resource limits. Other attributes that may influence scheduling decisions can be potentially added but we restrict to these three policy attributes in our study.

The policy attributes identify a set of categories to which jobs in a resource state will be assigned. How to represent and compare categorized resource state attributes becomes a central issue. We define a

new attribute type called numeric vector (“numeric-V”), which contains a vector of <key, value> pairs. The key contains values of selected policy attributes and represents a specific category. The value is a numeric scalar variable associated with that category. For example, let us assume that the selected policy attributes are group name and queue name (written as “group-queue”). They generate categories such as “bioinfo-qlong” and “astronomy-qshort”. Attribute “VRunJobs” contains a vector of pairs like <“bioinfo-qlong”, 32>, <“astronomy-qshort”, 8>, etc, where the value in each pair represents the number of running jobs in the category identified by the key. The same representation holds for other resource state attributes: “VQueueJobs” contains categorized number of queued jobs. “VAlreadyRun” (“VRunRemain”) is calculated as the number of CPUs multiplied by the elapsed (remaining) run time of the running jobs. “VAlreadyQueue” and “VQueueDemand” is computed by multiplying the number of CPUs with the already queued time or estimated run time, respectively. More formal definitions for resource state attributes are listed in Table 2. As we can see, different attributes are different views to describe a resource state. By introducing policy attributes we partition the resource states into a more fine grained level. Distances between these attributes are calculated using the distance function described in the next section.

4. The IBL-Based Prediction Algorithm

Instance Based Learning [2] techniques typically store training data in a historical database, and make predictions for a particular query by applying an induction model on its “nearby” data entries. Two main components of IBL techniques, namely the distance function to measure “nearness” and the induction models, are defined for our algorithm.

4.1. The Distance Function

We employ the Heterogeneous Euclidean-Overlap Metric (HEOM) [17] as the distance function. This distance function can be used on nominal and numeric scalar attributes, and we extend it so that it can also be used for numeric vector attributes. The extended HEOM distance function defines distance be-

tween two values x and y of a given attribute a as

$$d_a(x, y) = \begin{cases} \text{overlap}(x, y), & \text{if } a \text{ is nominal,} \\ ns_diff_a(x, y), & \text{if } a \text{ is numeric,} \\ nv_diff_a(x, y), & \text{if } a \text{ is numeric-V,} \\ 1, & \text{otherwise.} \end{cases} \quad (1)$$

The function *overlap* for nominal values is

$$\text{overlap}(x, y) = \begin{cases} 0 & \text{if } x = y, \\ 1 & \text{otherwise.} \end{cases} \quad (2)$$

The function *ns_diff* for numeric values is

$$ns_diff_a(x, y) = \frac{|x - y|}{max_a - min_a}, \quad (3)$$

where max_a and min_a are the maximum and minimum observed value for attribute a .

The function *nv_diff* for numeric vector values is

$$nv_diff_a(x, y) = \frac{\sum_{i=1}^{N_a} |x_i - y_i|}{range(a)}, \quad (4)$$

where i is the i th category and N_a is the total number of categories of x and y . $range(a)$ is the maximum difference of all categories observed in the training data for attribute a .

The above definition for d_a returns a value in the range from 0 to 1. Unknown attribute values are handled by returning a distance of 1, i.e., the maximal distance. The distance between two input vector of attributes \mathbf{x} and \mathbf{y} is given by the Heterogeneous Euclidean-Overlap Metric function $D(\mathbf{x}, \mathbf{y})$

$$D(\mathbf{x}, \mathbf{y}) = \sqrt{\frac{\sum_{a=1}^m w_a \times d_a(x_a, y_a)^2}{\sum_{a=1}^m w_a}}, \quad (5)$$

where w_a enables weighting for attributes.

4.2. The Induction Models

Two induction models, namely Weighted Average (n-WA) and Linear Locally Weighted Regression (n-LLWR) are considered in our algorithm.

n-WA: Weighted Average makes predictions using the weighted average of the nearest n neighbors

$$P(q) = \frac{\sum_{i=1}^n W_i \times Val(e_i)}{\sum_{i=1}^n W_i}, \quad (6)$$

where $W_i = K(D(q, e_i))$, $K(d) = e^{-(d/k)^2}$, q is a query, $P(q)$ is the value to be predicted, e_i is the i th nearest neighbors and $Val(e_i)$ is its target value, D is the distance function mentioned above, K is the Gaussian kernel function and k is the kernel bandwidth.

n-LLWR: A weighted linear model is fitted to the nearest n neighbors. Follow the formula in [2], the regressor is of the form

$$P(q) = q^T (Z^T Z)^{-1} Z^T v, \quad (7)$$

where $Z = WX$, and $v = Wy$. The weights are the squared root of the kernel function used in Equation (6), namely, $w_i = \sqrt{K(D(x_i, e_i))}$. W is a diagonal matrix with diagonal elements $W_{ii} = w_i$ and zeros elsewhere. X is the matrix form of attribute vectors of nearest neighbors and y is the target values. For details about this model we refer to [2].

5. Genetic Search

There are many parameters to be tuned in the basic IBL prediction algorithm. For instance, attribute weights have to be determined. The policy attributes have to be discovered for representing the local scheduling policies. The induction model, history database size, and the kernel bandwidth have to be set for a practically useful algorithm. We exploit a genetic search to optimize these parameters by minimizing the average prediction error on the training data. Our genetic algorithm uses real value encoding and it is designed using standard operators such as selection, mutation, and crossover [5]. Chromosomes are structured to match the different objectives, namely, application run time or queue wait time.

For application run times, the chromosome is structured as follows:

$\{ (w_g, w_u, w_q, w_e, w_n, w_r, w_{tod}), (\#CPUs), (method), (neighbor\ size), (history\ size), (bandwidth\ type), (bandwidth) \}$

The first block is the weights for job attributes, ranging from 0 to 1. The second block consists of attributes used for regression, in which the number of CPUs is the selected attribute. The third block is the induction model with two choices available: WA or LLWR. Other parameters are mostly self-explanatory. We evaluate two types of bandwidth selection, namely, ‘‘global’’ and ‘‘nearest neighbor’’. If global bandwidth is enabled, the value of the last

block is used as the fixed bandwidth in the algorithm. In nearest neighbor bandwidth selection, bandwidth k is set to be the distance to the n th nearest data entry. The crossover operator cuts between blocks so good patterns inside one block (like attribute weights) are not deconstructed. The genetic search is performed on the training set and attempts to minimize the average prediction error for application run times. The optimized parameters are then used for predictions on the test set.

For queue wait times the chromosome is:

$\{ (wp_g, wp_u, wp_q), (wa_g, wa_u, wa_q, wa_e, wa_n, wa_r, wa_{tod}), (ws_{rj}, ws_{qj}, ws_{alrr}, ws_{alrq}, ws_{rrem}, ws_{qdem}), (\#CPUs, queue\ demand\ credential, queue\ demand\ total), (method), (neighbor\ size), (history\ size), (bandwidth\ type), (bandwidth) \}$

Compared with that of application run times, the chromosome of queue wait times has two more building blocks. They are the weights for three policy attributes and six resource state attributes (see Table 2). Weights for policy attributes are binary to enable policy selection. Weights for resource state attributes are real values from 0 to 1. There are also two elements added to the regression attribute list. *queue demand credential* is the corresponding category value of ‘‘VQueueDemand’’ as identified by the query point. *queue demand total* is the sum of values in all categories for ‘‘VQueueDemand’’. These are potentially useful attributes that can be used in regression. As a simple example, the waiting time of a job would have a linear correlation with the total resource demand in a FCFS queue. During the test stage, two resource state attributes of queries, namely VRunRemain and VQueueDemand, are calculated using the estimated run times since the real values are not known at the prediction time. By adding application run time and queue wait time estimates we obtain predictions for response times.

6. Empirical Evaluation

We empirically evaluate our algorithm using real workload traces with different characteristics (Table 3). The NIKHEF cluster is a representative site in the LHC Computing Grid, which is primarily used for high energy physics data processing. It consists of a mix of Intel and AMD CPUs with different speeds, 512MB to 1GB memory, and Ethernet connections.

Name	Arch.	Batch System	Scheduler	CPUs	Period	#Jobs
NIKHEF	PC cluster	openPBS	Maui	288	Jun'04 - Dec'04	161666
SDSC01	IBM SP	LoadLeveler	Catalina	1152	Jan'01 - Dec'01	88694
SDSC02	IBM SP	LoadLeveler	Catalina	1152	Jan'02 - Dec'02	91751

Table 3

Characteristics of workload traces used in the experimental study. The NIKHEF and the SDSC traces are made publicly available via [11] and [12], respectively.

Name	Run Time		Wait Time		Response Time	
	Abs. Error	Nor. Error	Abs. Error	Nor. Error	Abs. Error	Nor. Error
NIKHEF	324.6 min	0.58	299.3 min	0.73	560.5 min	0.57
SDSC01	35.9 min	0.49	376.7 min	0.89	391.4 min	0.79
SDSC02	50.1 min	0.51	690.2 min	0.70	705.7 min	0.65

Table 4

The overall average absolute errors (Abs. Error) and normalized absolute relative errors (Nor. Error) of the algorithm in predicting run times, wait times, and response times.

The cluster is running Maui [9] as the local scheduling engine. Firstfit backfilling is employed in the scheduling algorithm. Priority/fairshare/throttling policies are enforced for different groups and users. Multiple queues are defined for jobs with different requested run times. The SDSC Blue Horizon is a terascale IBM SP supercomputer. The scheduler on this machine is called Catalina. It uses multiple submission queues with different QoS requirements, maintains one priority-based execution queue, performs backfilling, and supports reservations. Catalina is different compared with Maui primarily in that Catalina does not support fairshare and its policies are based on queues. The selected traces are suitable for experimental study because of the diversities in application types, machine architecture, and scheduling policies.

Two kinds of metrics are used for measuring the performance of our algorithm. *Prediction accuracy* is measured by the average absolute prediction error, defined as $\sum_{i=1}^N |t_{est} - t_{real}|/N$. t_{est} and t_{real} are predicted and actual values, respectively. We also show the normalized average absolute errors by dividing the average run time (or wait time, or response time). Moreover, the relative error is formulated as $r_e = (t_{est} - t_{real})/(t_{est} + t_{real})$. *Prediction time* is measured as the average execution time in milliseconds (ms) per prediction.

The evaluation is carried out on multiple Intel Xeon

machines with four 2.8GHz CPUs and 3GB shared memory. The workload dataset is divided into training and test sets throughout the evaluation. On NIKHEF, performance is evaluated on the test trace data (from August to December, 2004) of every one month, with the IBL parameters found by the genetic search on the preceding two-month traces. On the two one-year SDSC traces, testing is done on data from July to September and from October to December. The IBL parameters are optimized using the preceding six-month traces. The use of independent test sets enables an objective evaluation and a reflection of how search and predictions would be performed in practice.

Table 4 shows the prediction accuracy of our algorithm in terms of average absolute errors. As we can see, jobs from NIKHEF have longer run times but relatively shorter wait times than those from SDSC traces. Better results are achieved for run time predictions compared to wait times. If we combine the two, the response times have normalized average errors ranging from 0.57 to 0.79. Average results may be dominated for those with large values. For a closer look of how predictions are related to real values, histograms of relative errors r_e are plotted in Figure 1 and 2. As we can see, good accuracy is achieved for run time predictions in general. A majority of jobs have relative errors between -0.5 and 0.5, with the largest percentage centered around zero. The rela-

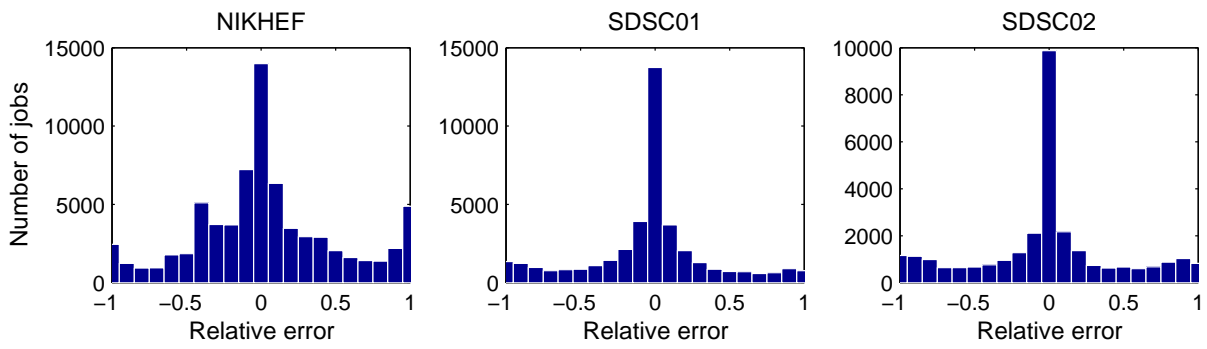


Figure 1. Histograms of relative errors for predicting application run times on three traces.

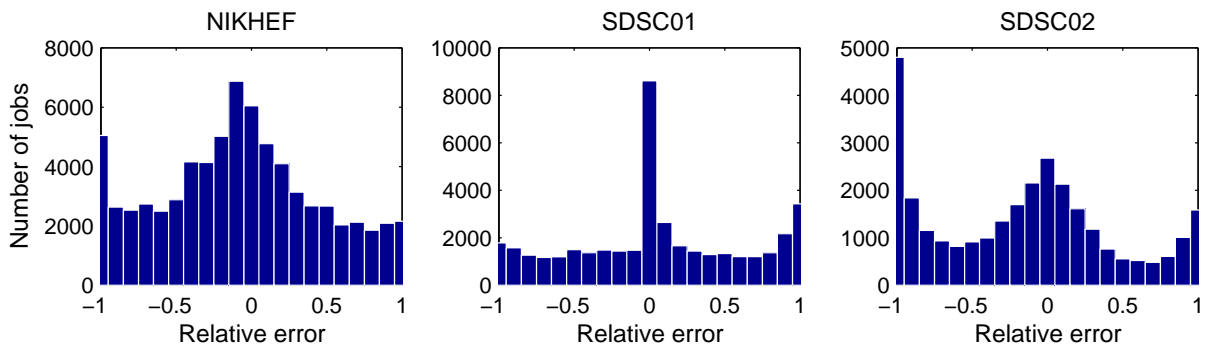


Figure 2. Histograms of relative errors for predicting queue wait times on three traces.

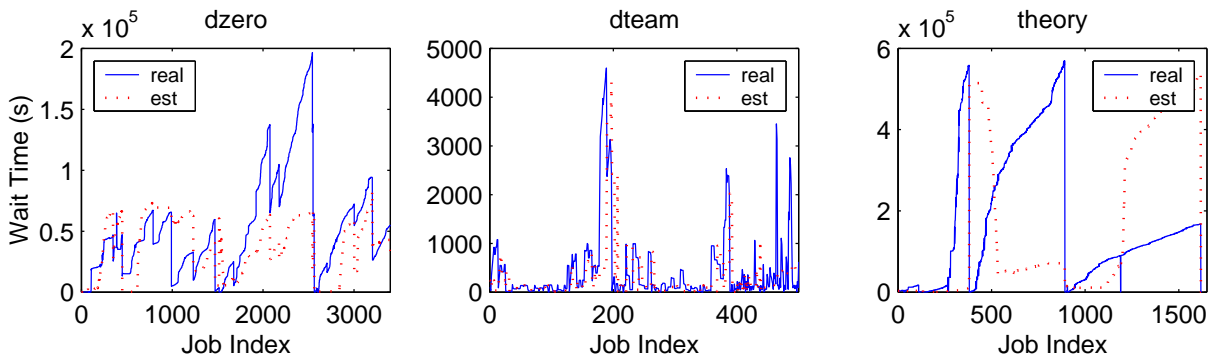


Figure 3. Comparison of real and estimated queue wait times for three representative groups on NIKHEF.

Name	Run Time (nocache)		Run Time (cache)		Wait Time	
	mean	std	mean	std	mean	std
NIKHEF	38 ms	28 ms	10 ms	8 ms	313 ms	185 ms
SDSC	30 ms	32 ms	23 ms	17 ms	461 ms	516 ms

Table 5

Mean and standard deviation (std) of average execution times for cached and not cached run time predictions, and wait time predictions. Caching interval $\Delta t = 100$ seconds.

tive errors on NIKHEF are more spreading from the center area compared to SDSC counterparts. This is partially because the NIKHEF cluster consists of heterogeneous nodes with different speeds whereas SDSC supercomputer has homogeneous processors. The node information is not known at the prediction time for queued jobs, resulting in larger errors for run time predictions. For queue wait times, the distributions of relative errors shown in Figure 2 are flatter and percentages of “bad” predictions increase. It indicates that the algorithm is less effective in predicting wait times than run times. This finding is expected since queue wait times are generally much more difficult to predict, involving dynamic resource states and more uncertain factors.

It is important that predictions should at least capture the trends and “categories” of real wait times. To further investigate the effectiveness of our wait time predictions, we evaluate three representative groups on the NIKHEF cluster. It is interesting to study group behavior on this site since most of the priority, throttling, and fairshare policies are defined based on groups. Figure 3 shows the comparisons of real and predicted values for the three groups. Group *dzero* has relatively high priority and usually submits massive similar data-parallel jobs in bags. We can see its waiting patterns is repetitive and quite predictable. Our technique is able to perform quite well for these jobs. We observe that the algorithm cannot follow the peak in the middle and this is because it is a new pattern never seen before. For group *dteam* which has many short-running test jobs, the waiting times may still be small even when the system is heavy-loaded and has a long queue. This is because of the backfilling and throttling policies. Our technique works well in this case while load-based predictors would most likely fail. For group *theory*, however, the predictions are not able to match the real values. The reasons are

three-folds: firstly, *theory* jobs are highly irregular and have large variances in run time predictions, which in turn result in big errors in estimating resource state attributes. Secondly, group *theory* has a relatively low priority and strict resource limits, whose waiting times are more easily affected by later coming jobs. Thirdly, there is not enough similar jobs to learn from in the history for group *theory*. An interesting observation in the plot of group *theory* is that predictions reproduce the previous pattern as the best effort, which fails to match the current trend. This reflects the poor performance of nearest neighbor learning when patterns shift. To sum up, although groups like *theory* may contribute considerably to the absolute average prediction error, they only represent a small fraction of all jobs. Our algorithm is able to perform for the most active groups thus also for a majority of jobs.

We now focus on the execution time of our algorithm. As mentioned above, jobs on NIKHEF typically arrive in “bags”. Namely, many similar jobs are submitted within short time intervals. On SDSC same patterns occur, although more distributed and less frequently. In this case caching can be adopted to reduce execution times. The estimation for a given attribute vector at time t is cached for another Δt seconds so jobs arrive before $t + \Delta t$ can use the same prediction. The caching mechanism is not used for wait time predictions as the resource state attributes may change quickly over time. Table 5 lists the average execution times for run time and wait time predictions. We can see that with caching the performance for run time predictions is improved significantly. The average execution time reduces almost 75% on NIKHEF. On SDSC, however, performance is improved only marginally. These are the results expected from their different arrival patterns. For wait times, SDSC also has longer and more spreading execution times. This is because the queue composition

on SDSC is more diverse than on NIKHEF, which results in more computationally expensive distance calculations. A large number of nearest neighbors and a big history size could also lead to worse performance. Nevertheless, with an average execution time of less than half a second, our algorithm performs much better than the simulation approach [7]. This is one of the main advantages of the IBL algorithm which makes it practically useful in a production Grid environment.

7. Conclusions and Future Work

Metascheduling in the Grid needs dynamic information about the resources. In this paper we proposed an Instance Based Learning technique to predict job response times using historical workload data. We define a new type of attributes (numeric vector) to represent a resource state, and introduce a distance function that can measure distances between these attributes. Local scheduling policies are embedded in the resource state attributes and they can be automatically discovered using the genetic search. This is an important step towards a more general solution.

An extensive empirical evaluation is conducted using traces with diverse characteristics. Generally speaking, our algorithm achieves acceptable prediction accuracy. The absolute relative errors for response time predictions range from 0.57 to 0.79. Detailed analysis based on representative groups on NIKHEF shows both the strength and weakness of our technique, and for a majority of jobs the technique is able to work quite well. With the average execution time in the order of milliseconds, the algorithm performs much better than the scheduler simulation counterparts. This yields a practical solution towards response time predictions, which serves as real-time information to support metascheduling decisions.

Our technique has its limitations as well. A reasonable amount of data is needed by the genetic algorithm for optimizing parameters. The evaluation process could take a long time if the execution time per prediction increases. Predictions for certain jobs are being highly inaccurate. Therefore, we are improving our technique in several directions. Firstly, we are investigating a local tuning method that tunes parameters for each policy group. The evaluation phase can be accelerated by parallelizing tuning processes. The prediction accuracy can also be potentially improved as pa-

rameters are fitted for each target group. Secondly, the nearest neighbor search in the current algorithm is sequential, which leaves much space for improvement. We are experimenting different search tree structures to speed up searching process [4]. Thirdly, we are developing a toolkit called PDM (Performance Data Miner [11]), which includes implementations of algorithms proposed in this paper. We are also investigating metrics to reflect prediction inaccuracies and algorithms for resource brokers that can incorporate inaccuracies when making decisions.

Acknowledgments

We thank Jeff Templon (NIKHEF) for his help and discussions. We also want to express our gratitude to the Parallel Workload Archive through which the SDSC traces are made publicly available. We are grateful to the reviewers for their many valuable suggestions that improved the quality of this paper.

REFERENCES

1. D. Abramson, R. Buyya and J. Giddy. A computational economy for grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems*, 18(8):1061–1074, 2002.
2. C. G. Atkeson, A. W. Moore, and S. Schaal. Locally weighted learning. *Artificial Intelligence Review*, 11(1-5):11–73, 1997.
3. J. Cao, D. P. Spooner, S. A. Jarvis and G. R. Nudd. Grid load balancing using intelligent agents. *Future Generation Computer Systems*, 21(1):135–149, 2005.
4. E. Chavez, G. Navarro, R. Baeza-Yates, and J.L. Marroquin. Searching in Metric Spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.
5. D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Inc., 1989.
6. N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley. Predictive application-performance modeling in a computational grid environment. In *proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, IEEE Press, 1999.
7. Hui Li, David Groep, Jeff Templon, and Lex

- Wolters. Predicting job start times on clusters. In proceedings of *4th IEEE/ACM International Symposium on Cluster Computing and the Grid*, IEEE Press, 2004.
8. Hui Li, David Groep, and Lex Wolters. Efficient Response Time Predictions by Exploiting Application and Resource State Similarities. In proceedings of *the sixth IEEE/ACM International Workshop on Grid Computing*, IEEE Press, 2005.
 9. The Maui Scheduler. <http://www.supercluster.org/maui/>.
 10. Jarek Nabrzyski, Jennifer M. Schopf, and Jan Weglarz (Editors). Grid Resource Management: State of the Art and Future Trends. ISBN: 1402075758, Springer, 2003.
 11. PDM - A Performance Data Miner. <http://www.liacs.nl/home/hli/pdm/>.
 12. Parallel Workload Archive. <http://www.cs.huji.ac.il/labs/parallel/workload/>.
 13. W. Smith, I. Foster, and V. Taylor. Predicting application run times using historical information. *Lecture Notes in Computer Science*, 1459:122–136, 1998.
 14. W. Smith. Resource Management in Metacomputing Environments. PhD thesis, Northwestern University, 1999.
 15. W. Smith, V. Taylor, and I. Foster. Using runtime predictions to estimate queue wait times and improve scheduler performance. *Lecture Notes in Computer Science*, 1659:202–219, 1999.
 16. C. Weng and X. Lu. Heuristic scheduling for bag-of-tasks applications in combination with QoS in the computational grid. *Future Generation Computer Systems*, 21(2):271-280, 2005.
 17. D. R. Wilson and T. R. Martinez. Improved heterogeneous distance functions. *Journal of Artificial Intelligence Research*, 6:1–34, 1997.