UNIVERSITY OF CALIFORNIA
Santa Barbara

Statistical Methods for Mitigating Resource Provisioning Dynamism in Large-Scale Batch-Scheduled Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Daniel C. Nurmi

Committee in Charge:

Professor Rich Wolski, Chair

Professor Amr El Abbadi

Professor Ben Zhao

March 2009

The Dissertation of
Daniel C. Nurmi is approved:

Professor Amr El Abbadi

Professor Ben Zhao

Professor Rich Wolski, Committee Chairperson

March 2009

Statistical Methods for Mitigating Resource Provisioning Dynamism in

Large-Scale Batch-Scheduled Systems

Copyright © 2009

by

Daniel C. Nurmi

Curriculum Vitæ

Daniel C. Nurmi

Education

2002	Master of Science in Computer Science, University of Chicago.

Experience

2002 – 2008	Graduate Research Assistant, University of California, Santa Barbara.
Winter 2003	Associate Lecturer, University of California, Santa Barbara.

Selected Publications

D. Nurmi, J. Brevik, R. Wolski, "VARQ: Virtual Advance Reservations for Queues" *ACM/IEEE International Symposium on High Performance Computing (HPDC)*, June 2008.

D. Nurmi, J. Brevik, R. Wolski, "Queue Bounds Estimation from Time Series" *13th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2007.

D. Nurmi, J. Brevik, R. Wolski, "Probabilistic CO-Allocation Over Best Effort Batch Scheduled Resources" (to appear) *Cluster Computing Special Issue on High Performance Distributed Computing*, 2009.

Abstract

Statistical Methods for Mitigating Resource Provisioning Dynamism in Large-Scale Batch-Scheduled Systems

Daniel C. Nurmi

Users of high performance computing (HPC) systems generally rely on concurrency to achieve performance. Modern users have the ability to draw from a vast array of distributed resources due to the ever increasing quality of connecting software and networks. However, as the pool of resources available to users grows, so does the level of resource heterogeneity and performance response dynamism.

Historically, users request access to a super-computer's resources by submitting their work and waiting until the system has enough free resources to satisfy the user's request. However, few facilities exist that cater to the substantial class of users who require that their work is completed by a specific time, who require that their resources are available during a specific time interval, or who require simultaneous access to multiple systems.

In this dissertation, we discuss new statistical methodologies to manage resource performance dynamism, and abstractions that build upon these methodologies to hide resource heterogeneity. In particular, we will show how we have successfully developed the methodologies and abstractions necessary to manage and hide provisioning delay of HPC resources.

Contents

Curriculum Vitæ **iv**

Abstract **v**

List of Figures **viii**

List of Tables **ix**

1 Introduction **1**

2 Problem Statment **6**

3 Approach **13**
3.1 Data 19

4 Batch Queue Delay Prediction **23**
4.1 Introduction 23
4.2 Methodology 27
4.2.1 Quantile Prediction 28
4.2.2 History Trimming 32
4.2.3 Job Clustering 35
4.2.4 Model-Based Clustering 36
4.2.5 Availability Inference 41
4.3 Experiments and Results 43
4.3.1 Simulation 45
4.3.2 Correct and Accurate Predictions 47
4.3.3 Experiments 49
4.3.4 Correctness Analysis 51

4.3.5 Accuracy Analysis 56
4.4 Conclusions 59

5 Virtual Advance Reservations 63
5.1 Introduction 63
5.2 Methodology 67
5.2.1 Virtual Advanced Reservations 68
5.3 Experiments and Results 80
5.3.1 Efficacy Experiments and Apparatus 81
5.3.2 Efficacy Results 86
5.3.3 Allocation Overhead Results 88
5.3.4 Generality Experiments and Apparatus 90
5.3.5 Generality Results 92
5.3.6 Discussion 94
5.4 Conclusion 97

6 Statistical Co-allocation 98
6.1 Introduction 98
6.2 Methodology 100
6.2.1 CO-VARQ Overview 103
6.3 Experiments and Results 108
6.4 Conclusion 111

7 Related Work 113
7.1 Backgound and Related Work 113
7.1.1 Resource Availability Dynamism 113
7.1.2 Resource Provisioning Dynamism 116
7.1.3 Co-Allocation of Distributed Resources 120

8 Impact and Conclusion 124
8.1 Impact 124
8.2 Conclusion 126

Bibliography 129

List of Figures

3.1 Example queue delay time-series. 15

4.1 Job delays and 0.95 quantile predictions on CNSI Dell. 53
4.2 Linear delay increase property with 0.95 quantile predictions. . . . 54
4.3 Comparison of log-normal and binomial prediction methods. . . . 55

5.1 Overview of VARQ/system infreastructure interaction. 68
5.2 Job delays and 0.95 quantile predictions on Datastar. 72
5.3 VARQ algorithm pseudocode. 76
5.4 Probability trajectory on NCSA TeraGrid. 79

6.1 Example CO-VARQ procedure. 103

List of Tables

3.1 HPC Resource description. 20
3.2 Job trace data summary statistics. 21

4.1 BMBP empirical experiment results. 61
4.2 QBETS experiment empirical results. 62

5.1 Resources used for VARQ experiments. 78
5.2 VARQ empirical experiment results for minimum success probability of 0.50 82
5.3 VARQ empirical experiment results for minimum success probability of 0.75 82
5.4 VARQ empirical experiment results for minimum success probability of 0.95 83
5.5 VARQ experiment over-allocation costs 89
5.6 VARQ simulation experiment results 92

6.1 Resources used for CO-VARQ experiments. 108
6.2 CO-VARQ experiment empirical results. 110

Chapter 1

Introduction

Computational scientists from all disciplines depend on large-scale, high performance computing (HPC) and data storage resources to further their science. Modern scientific applications vary in structure, but for the most part share in common the need for a variety of resources, including large collections of compute resources connected via high performance networks, complex data storage and movement facilities, specialized rendering and visualization systems, and special purpose scientific instruments. While many applications require one or few general types of resources to execute, an increasing number draw from as many available resources as possible, during the lifetime of the application, in order to achieve high performance.

Typically, high performance resources are provisioned and controlled by individual organizations located at top class Universities, National Laboratories, and other Government funded organizations. While many of these organizations

promote an open policy when it comes to granting access of the resources to the scientific community, they typically each manage and configure local resources independently, without adhering to any centralized policies or resource management schemes. There are many reasons why HPC resources are managed in this way. First, the hardware and software advances that today comprise HPC systems changes so rapidly that maintaining a centralized management scheme would hobble the ability to keep our high performance systems up-to-date, since keeping independently managed, distributed HPC resources homogeneous is infeasible. Second, there is significant anecdotal evidence that resource heterogeneity promotes experimentation and discovery (the successful transition from HPC centers primarily hosting specialized super-computing resources to HPC systems composed of large collections of commodity components being a good example).

As the number of high performance resources has increased and interconnecting networks improved dramatically in speed, computational scientists and systems engineers alike became increasingly interested in using multiple distributed high performance resources simultaneously but, due to the lack of central management, this task proved to place a substantial burden on users. Thus, in the mid-to-late 1990s, the concept of Meta-computing (later termed Grid-computing) was introduced, based around the idea that individual resources would remain individually managed as long as standardized interfaces existed between them, allowing global

entities, be they users or centralized resource managers, to control large-scale, federated collections of high performance resources. The vision of Grid-computing, if realized, allows a computational scientist to easily develop an application and then draw computational and data power from the "computational grid" just as our day-to-day appliances draw electrical power from the "power grid". There have been significant strides made toward achieving this goal, but many challenges still remain before the vision can be truly realized.

In this dissertation, we identify and provide a solution to a significant problem that, although stated as a first class requirement in principle Grid-computing literature [10], has remained unsolved for over a decade; methodologies for providing time sensitive resource provisioning services. Most HPC sites today require scientists to formulate requests for resources in the form of a *job*, where the specifics of required resources are specified along with instructions for how to execute the scientist's application. Due to the fact that HPC resources are commonly overcommitted, these job requests are often queued until the requested resources become available. We observe that the amount of time that passes between when a job is queued and when it is executed is a highly variable and can be a substantial fraction of the overall turnaround time of the job (queue time plus the actual application run time). Job queue delay uncertainty blocks scientists from effectively planning their experiments, and is a major obstacle to users who need

to place strict time requirements on the execution of their applications. Thus, the question that we address in this dissertation is;

Is it possible to predict and hide, through abstraction, provisioning delay variability in large scale HPC systems?

Our approach to this problem is to first gather and analyze historical data of observed job queue delays, use this data to make predictions on the future queue delay of individual jobs, and to finally create abstractions that are familiar to scientists that use these predictions in order to provide two services that are currently unavailable to the general HPC community. We have developed a methodology for predicting, in real time, bounds on future job delay that has been implemented as a general prediction service. Atop this service, we have built an abstraction for provisioning resources at a single HPC site during a specified time interval, and another abstraction for provisioning resources at multiple sites simultaneously. Thus, the main contributions of this work are as follows;

- We provide a new statistical methodology for predicting probabilistic upper bounds on resource provisioning delay, which plays a significant role in determining overall HPC application turnaround time.

- Using the methodology, we provide two new abstractions that provide HPC users and tools with probabilistic services for provisioning resources with strict time constraints.

In the next Chapter 2, we concretely define the problem that this dissertation will address. In Chapter 3, we describe our general approach to the problem. Chapter 4 details our prediction methodology and results, followed by Chapters 5 and 6, which describe two new abstractions and experimental verification of their effectiveness on HPC systems in operation today. Finally, we conclude in Chapter 8 and discuss related literature in Chapter 7.

Chapter 2

Problem Statment

Most production HPC centers serving the scientific and engineering research communities use space-sharing [19] to manage the allocation of compute and data resources to user programs. Users run their programs on a given HPC machine by submitting them to a "batch scheduler" as textual representations, each specifying a program to be run and its particular resource requirements. When executed, each program (termed a "job") is given exclusive access to a partition of the machine for its execution duration. Thus jobs share the *space* of available processors within a machine, but each processor is not time-shared among competing programs submitted by multiple users.

Space sharing (as opposed to time sharing) maximizes the efficiency of resource usage when a parallel program is executing [28, 36, 16, 26]. However, to be granted exclusive access to a resource partition, user jobs are managed in *batch mode*. That is, they are submitted to a batch scheduler and then held in a queue

until a sufficient number of processors become available to run the job at the head of the queue. The queue-scheduling discipline implemented in a production setting is rarely, if ever, as simple as first-come-first-served (FCFS). Instead, system administrators control scheduling priority through a policy interface, specifying how jobs waiting for execution should be chosen when sufficient resources within the machine become available. These policy specifications, in practice, are quite complex, site-specific, and dynamic, since they must balance user experience (job turnaround time, fairness) with changing site priorities (project demonstrations, paper deadlines for important users, etc.) while at the same time keeping overall resource utilization high. Moreover, even when there is an attempt to make site management software loosely globally synchronized, each machine has its own distinct runtime scheduler algorithm and set of ever-changing policies. That is, the administrative mechanisms governing these systems are as heterogeneous and dynamic as the workload experienced by the systems themselves.

In most of these settings, each user (represented by a unique per-site user identifier) is associated with one or more accounts each funded with an allocation of per-node occupancy time. When a user submits a job to the local batch scheduler, they must specify which account to charge when the program is eventually allocated machine resources for execution (on some systems, the scheduler may choose a default for the user if none is specified). There is no charge made to the

account while the job is waiting in queue, nor is any form of refund or compensation granted for jobs that wait for overly long periods. It is only the execution occupancy time that is decremented from the account once the program begins executing. If the account is exhausted, the currently executing jobs charging the account are terminated. Note that in most settings the "aspect ratio" of a parallel job submitted by a user is specifically not considered in the accounting subsystem. That is, a 1-node job executing for 100 hours decrements the user's account by the same quantity that a 100 node job executing for 1 hour does: 100 node-hours.

Typically, site administrators configure their batch systems to employ a simple fundamental scheduling policy based on techniques such as first-come-first-serve (FCFS), and then perform further tuning, taking into consideration specific job and/or user priority goals that are unique to each site. A nice overview of current parallel job scheduling techniques is provided in [21] and a more comprehensive survey for past methods is provided in [17]. In particular, most sites currently use some form of backfilling [34, 40, 45] to maintain utilization in the presence of large resource requests without introducing starvation. Utilization, in this context, is measured in terms of node occupancy. Once a user's job is allocated a set of nodes, the system does not (and almost certainly cannot) determine whether the work done by the program is useful – only that an account should be charged for the occupancy. To simplify this accounting process, and also to ensure that a

scientist's application is not being effected by other work in the system, queuing policies do not allow job pre-emption (once a job is granted access to resources, no other job can interrupt that job) or a general facility for application checkpointing (jobs that start typically expect to run to completion).

If large node counts are needed by a job, the machine must "drain" until a sufficient number of nodes become available (recall that pre-emption and/or checkpointing is typically not available). To avoid the potential loss of utilization that results during a drain, each job can specify a maximum execution time, past which it is willing to be terminated. A backfilling scheduler will use these execution limits to schedule jobs from farther down in the queue onto draining nodes such that they do not prolong the waiting time of a job causing the drain that is ahead of them.

To give users some measure of predictability and control, many centers configure different queues with partially described scheduling priorities to allow users to make some form priority-motivated decision. For example, a "short" queue may accept jobs that have maximum run times no greater than 15 minutes which are given preference during backfilling. Users can use this information to gain faster turn-around for small amounts of work, but the exact degree of preference is typically not revealed, and local administrators change the specific policy parameters without announcement (sometimes frequently).

For these reasons, but also because large-scale parallel programs have widely varying execution times [8, 29] and also because HPC resources in research settings tend to be extremely overcommitted, user jobs experience highly variable queuing delays [4, 11, 12, 60]. At present, for example, queuing delay for jobs submitted at many production HPC centers may vary by several orders of magnitude [1]. This degree of variability is often of great concern to the general user population. Indeed, the queue delay experienced by a job may exceed (by several orders of magnitude in some cases) its execution time. As a result, users with very efficient and highly tuned programs may experience unexpectedly long turnaround times because their jobs had to wait in queue.

The degree of variability found in queue delay is the root cause of a number of significant problems that users face when attempting to plan and provision resources for application execution cycles. For instance, if a user has immediate access to a small number of local resources and access to much larger batch resources, it is not immediately clear which system would be able to complete the application faster from submit time to execution completion time. Even through the larger system will process the application much faster than the local resource once the resources are provisioned, the delay imposed by batch queues can outweigh the benefits of running on the larger, faster machine. Further, if the user

[1] For evidence, see Feitelson's workload archive at `http://www.cs.huji.ac.il/labs/parallel/workload`.

imposes a hard deadline by which their application must complete, variability in batch queue delay makes it implausible to assert, with any useful certainty, that a job will complete before a deadline. Another obstacle that batch queue delay creates becomes apparent when a user requires access to resources during a specific time interval. This occurs when applications are tied to some real-time event like gathering data from weather stations, when visualization resources must be utilized to steer an application during it's execution, or in any other case where the user must plan in advance the availability of resources as part of their application. Finally, batch queue delay prevents users from co-allocating, or provisioning multiple sets of resources at different sites. Indeed, in [10], which outlines basic technical requirements that need to be addressed before the vision of Grid-computing is fully realized, the authors point out that a resource co-allocation service must exist, but for over a decade such a service has remained unavailable to the general HPC community.

Predicting the time an individual job will wait in a given queue when it is submitted has been the subject of previous research [11, 12, 60] but reliable and accurate predictions of the delays experienced by individual jobs (until recently) have remained elusive. Thus users of applications that depend upon specific time constraints currently find it difficult or impossible to plan effectively to meet application deadlines. Specifically, what is needed is a methodology that allows users

to reason about job queuing delay (queue delay prediction), a technique for ensuring that they can acquire resources at a specific time in the future for a specific duration (advanced reservations), and a technique that allows the simultaneous provisioning of resources across sites (resource co-allocation).

Chapter 3

Approach

The need for a methodology that allows scientists to plan ahead is at the core of the problems discussed in the previous Chapter. Currently, scientists cannot plan ahead due to the large amount of variability in resource provisioning delay that they experience when trying to run applications at major HPC sites. Our approach seeks to first find a way to predict the amount of time that a future resource provisioning request will wait before the resources become available for use. While making an *exact* prediction on this time is desirable, previous efforts [11, 12, 60] indicate that providing a prediction of future expected delay (typically phrased as a prediction of the mean/average provisioning delay) results in a good summary prediction for all resource provisioning requests, but is not as useful as a prediction for individual requests. We observe that when a scientist has an application to execute, they are less interested in the provisioning delay of all jobs than they are for the delay their individual job will experience. However,

due to the highly variable nature of observed individual job delays, combined with imperfect knowledge of the underlying sources of this variance, making exact predictions on individual jobs remains extremely difficult. However, we observe that the ability to predict an accurate *upper bound* on the amount of time a job will be delayed is sufficient information for the scientists to plan ahead. While defining an absolute upper bound prediction is straightforward (say, the age of the universe), it is not useful to do so. We propose that assigning a probability to an upper bound prediction would allow a scientist to decide how likely it is that an upper bound prediction is correct. Our general approach, then, is to make a probabilistic upper bound prediction on the amount of time an individual job will wait in queue before execution.

For a given HPC site, we consider individual job delay times to be a random variable from a population with an unknown distribution. As jobs leave the batch queue, we can record experienced job delay times as observations from this population as a time series. Figure 3.1 is an example time series gathered from the University of Chicago/Argonne National Laboratory's TeraGrid super-computer over a one month time period. Each individual graph feature represents the time that a single job waited in queue before it began execution.

Our general approach is to use these observations to estimate the value from the population that is greater than all other values with some probability q. In

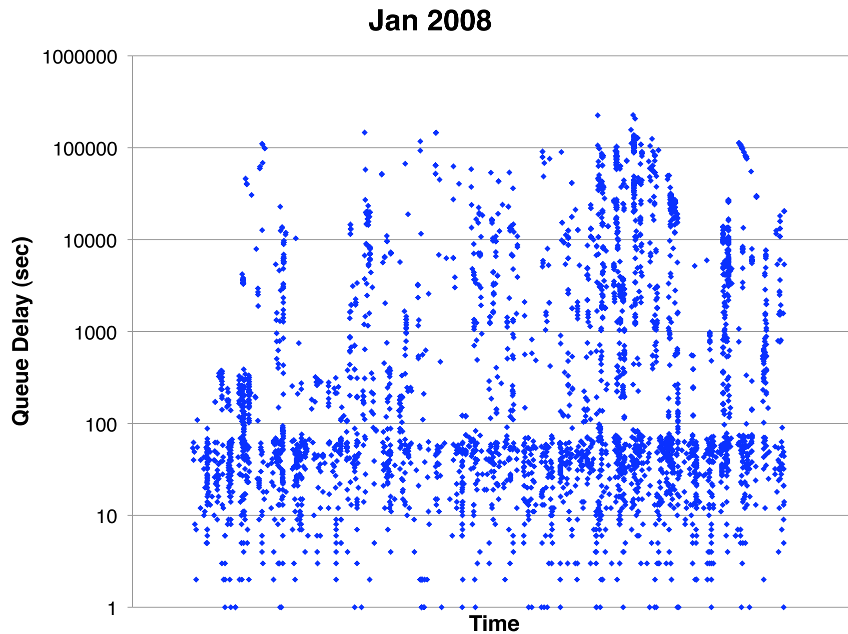

Figure 3.1: One month of batch job delay measurements from the UC/ANL TeraGrid system (ucteragrid). Each graph feature represents the number of seconds that a job waited in queue before it began execution.

other words, we seek to estimate given population's *q quantile* and use it as a probabilistic bound prediction for future observations drawn from the population. For example, if we knew the 0.50 quantile (median) value of the population from which the observations shown in Figure 3.1 were drawn, we could accurately predict that the next job will be delayed at most the median amount of time, 50% of the time (the other 50% of the time, the job would wait more than that amount of time). This would give us an upper bound delay prediction with a 0.50 probability of being correct. If we required a more certain upper bound prediction, we could

use higher quantiles (0.90, 0.95, etc.). Our challenge is to find a technique that, given a historical trace of job delay observations and quantile of interest, allows us to accurately estimate the population quantile of interest for use as a predictor.

Our methodology depends on our ability to gather, in near real-time, historical traces of batch queue delay times as jobs are processed. Using empirical observations from these traces, we provide a prediction methodology that effectively makes probabilistic upper bound predictions on the future queue delay of individual jobs. Building on this technique, we present two novel abstractions (Advance Reservation and Resource Co-allocation services) that are implemented as probabilistic services.

This layered approach implements functionality that has been previously sought-after in production settings but heretofore has remained largely unrealized. For example, at present, among NSF HPC centers supporting users in production, only the San Diego Supercomputer Center (SDSC) supports user-settable advanced reservations. Even so, at the time of this writing, it does so only on a few of its machines, and only on an experimental basis, after approximately 14 years of internal evaluation and development. Our abstractions implement these services in an entirely new way that does not require modification of local scheduling policies or implementations. That is, this new functionality *overlays* existing batch-scheduling infrastructure and does not require the cooperation of the local

site administration to function. Indeed, local administrators cannot determine whether a specific user is using our system or is going through the normal local mechanisms for job submission.

Our approach makes it possible for each site to retain complete local control over its systems. Our methods automatically sense changes in local policy and adapt to the new regime shortly after it is implemented. Thus, the impact on the local administration and software base is virtually nonexistent. This low-impact approach is important because implementing software and policy changes across the sites uniformly has proved labor-intensive, time consuming and error-prone.

At the center of this approach is a new non-parametric time-series analysis method we developed to make predictions, with confidence intervals, that are meaningful even when based on a relatively small number of measurements and in the presence of such autocorrelation structures as are typical in batch queues.

Our prediction method further uses an automatic change-point detection mechanism and model-based statistical clustering to sense and adapt to changing local conditions (*e.g.*, scheduling policy). Using this predictive capability, we can adaptively schedule the submission of user jobs to arbitrary queues so that they execute in specific time slots (advanced reservations) on one, or more machines simultaneously (co-allocations).

Additionally, our approach addresses one of the major impediments to the development of advanced reservation and co-allocation facilities, namely their potential impact on machine utilization. It is not technically difficult to provide users with either advanced reservation or co-allocation reservation mechanisms by modifying the local scheduling policies. Rather, the problem is that reserving resources, even in conjunction with efficiency-enhancing scheduling techniques such as back-filling [34, 40, 45] or pre-emption and/or OS checkpointing [15], may lead to a significant loss of resource utilization. HPC centers often use resource utilization as a measure of user utility and thus also a measure of the degree to which the "up-front" cost associated with siting a machine has been amortized. For each solution we propose in this dissertation, we discuss the potential impact the techniques may have on machine utilization and find that, unlike the situation with "hard" reservations, this impact is minimal.

In the following Chapters, we discuss in detail our prediction methodology and two abstractions that have each been implemented as generally available services, as follows:

- **Q**ueue **B**ounds **E**stimation from **T**ime **S**eries (**QBETS**) is an on-line prediction technique and implementation that forecasts bounds batch queue delay experienced by individual jobs in real time.

- **V**irtual **A**dvanced **R**eservations for **Q**ueues (**VARQ**) is an advanced reservation system that uses QBETS to make predictions of queue delay which it then uses to schedule job submissions so that they meet user-specified deadlines.

- Finally, **CO-VARQ** is a co-allocation service that uses VARQ to make simultaneous reservations on multiple systems to enable cross-site execution.

In the next section, we briefly discuss the nature of the historical batch queue delay traces that we use to make bound predictions.

3.1 Data

Over the course of several years, we have obtained 11 archival batch-queue logs from different high-performance production computing settings covering a variety of machine generations and time periods. Table 3.1 displays summary information about each resource, along with a shorthand tag (listed in the *Machine* column of the table) that we use to refer to traces from that machine throughout this dissertation. Typically, HPC site administrators define several "queues" to which users can choose to submit a job. For all systems except the ASCI Blue Pacific system at Lawrence Livermore National Laboratory (LLNL), each queue determines, in part, the priority of the jobs submitted to it.

Machine	Processors	Batch Software	Description
datastar	2176	Load Leveler	SDSC IBM PowerPC Production Compute Cluster
ucteragrid	316	Torque/Maui	UC/ANL IBM/Intel Compute Viz Linux TeraGrid Cluster
dante	35	Torque/Maui	RENCI Intel Xeon Research Linux Cluster
cnsidell	256	Torque/Maui	UCSB NanoScience Research Linux Cluster
ncsateragrid	1744	Torque/Maui	NCSA IBM/Intel Compute Linux TeraGrid Cluster
iuteragrid	32	PBS	IU AVIDD Compute Linux Cluster
ornlteragrid	56	Torque	ORNL IA64 Compute Linux Cluster
lonestar	5840	LSF	TACC Dell Linux Cluster
sdscteragrid	524	Torque/Maui	SDSC IBM/Intel Compute Linux TeraGrid Cluster
tsubame	10368	SGE	Tokyo Institute of Technology AMD Compute Cluster
ctc	430	Unknown	Cornell Theory Center SP-2
llnl	336	Unknown	Lawrence Livermore National Laboratory SP2
sdscblue	1152	Unknown	SDSC Blue Horizon

Table 3.1: HPC machines from which data was gathered for trace-based simulation and empirical experiments.

Site/Machine	Job Count	Avg. Delay	Median Delay	Std. Deviation
iuteragrid	8055	3858	2	21125
cnsidell	12446	85190	172	193946
ornlteragrid	25920	568	1	9221
tsubame	45054	5722	5	65786
llnl	56028	19625	2001	49997
sdscteragrid	62426	34795	207	112305
dante	65203	7956	57	16126
ncsateragrid	67391	33235	1631	107449
ctc	77216	25541	2197	93617
datastar	102584	43764	1048	137098
lonestar	137577	8365	10	51378
sdscblue	149633	47716	2019	139789
ucteragrid	218366	5707	19	31323

Table 3.2: Batch job delay trace data. The units for the mean, median and standard deviation measurements are seconds.

For each site that is still in existance at the time of this writing, we have deployed a sensor that uses the Network Weather Service (NWS) [67] infrastructure to gather, in real time, new job data as workload continues to be processed at each site. Collectively, the data comprises 4.6 million job submissions spanning approximately a 12-year period.

Within each log, each job is represented uniquely by five values: the queue to which the job was submitted, the submission time, the queue wait time, the number of nodes requested, and the maximum number of execution seconds requested. In Table 3.2, we show summary statistics for each of the job traces used in this dissertation. We discovered that, although each of these systems falls un-

der the general definition of a “super-computer”, and many are architecturally similar in nature, the variability found in the amount of time jobs wait in queue before executing is quite high within each machine, and quite different between machines. While many attempts have been made to predict batch queue job wait time based upon summary statistics like those presented in Table 3.2, when we look at the statistics together, we note very quickly that while these statistics can be used to accurately summarize a large set of batch queue wait times, they are not necessarily accurate predictors for individual jobs. Five of the traces, for instance, exhibit a standard deviation over 100000 seconds (more than a day), indicating that real queue wait time will deviate far from the mean. This observation helps to explain why, to date, users and meta-schedulers find the problem of deterministically provisioning HPC resources a daunting task.

In Figure 3.1, we show an example of observed queue delays from a single machine/queue. Here, we show the actual number of seconds that jobs waited in queue on the UC/ANL TeraGrid machines (ucteragrid) during the month of January, 2008. Note that while many jobs waited in queue for a very short time (between 1 and 10 seconds), another significant set of jobs waited 100000 seconds or more (over a day) before they began execution. We find this high level variability in wait time to be a common characteristic in most of the job traces we've gathered over multiple time scales (days to years).

Chapter 4

Batch Queue Delay Prediction

4.1 Introduction

In this Chapter, we present a method for automatically predicting bounds, with quantitative confidence levels, on the amount of time an individual job will wait in queue before it is initiated for execution on a production "batch scheduled", space-shared resource. The method consists of three interacting but essentially independent components: a percentile estimator, a change-point detector, and a clustering procedure. At a high level, clustering is used to identify jobs of similar characteristics. Within each cluster, job submissions are treated as a time series and the change-point detector delineates periods of stationarity. Finally, the percentile estimator computes a quantile that serves as a bound on future wait time based only on history from the most recent stationary region in each cluster. All three components can be implemented efficiently so that on-line, real-time

predictions are possible. Thus, for each job submission, our method can generate a predicted bound on its delay using a stationary history of previous jobs having similar quantitative characteristics. In addition, as jobs complete their time in queue, new data becomes available. Our method automatically incorporates this information by adjusting its clustering and change-point estimates in response to the availability of new data.

The percentile estimation method we describe here is a product of our previous work in predicting the minimum time until resource failure [3, 47, 49]. In this Chapter, we describe its application to the problem of predicting bounds on the delay experienced by individual jobs waiting for execution in batch-controlled parallel systems. To do so effectively, we have coupled this methodology with a new method for detecting change points in the submission history and a new clustering methodology that automatically groups jobs into service classes. This latter capability is necessary since many sites implement dynamically changing priority schemes that use "small" jobs to "backfill" [40] the machine as a way of ensuring high levels of resource utilization. Moreover, our quantile-based prediction method makes it possible to infer when the the machine may have crashed while the queuing system still accepts jobs (a common failure mode in these settings where jobs are submitted from one or more "head" nodes). Using this new system, we have found that it is possible to predict bounds on the delay of

individual jobs that are tighter then parametric methods based on Maximum Likelihood Estimation (MLE) of Weibull, log-normal, and log-uniform distributions. To achieve these tighter bounds, however, all four components – non-parametric quantile estimation, change-point detection, clustering, and availability inference – must be integrated and employed in concert. Because the systems in inherently an adaptive time series forecasting methodology, we give it the name QBETS as an acronym for **Q**ueue **B**ounds **E**stimation from **T**ime **S**eries.

We compare QBETS with various parametric methods in terms of prediction correctness and accuracy. We also demonstrate how the combination of techniques that compose QBETS improves the predictive power for production systems.

Our evaluation uses job submission traces from 11 supercomputers (including 8 currently in operation) operated by the National Science Foundation and the Department of Energy over the past 10 years comprising approximately 1.4 million job submissions. By examining job arrival time, requested execution time, and requested node count, we simulate each queue in each trace and compute a prediction for each job. Our results indicate that QBETS (which is more effective than competitive parametric methods) achieves significantly tighter bounds on job wait time in most cases. Thus the system automatically "reverse engineers" the *effective* priority scheme that is in place at each site and determines what job sizes are receiving the fastest turn-around time.

Thus, the work presented in this Chapter makes two significant new contributions with regard to predicting individual job queue delays.

- We present QBETS as an example of an accurate, non-parametric, and fully automatic method for predicting bounds (with specific levels of certainty) on the amount of queue delay each individual job will experience.
- We verify the efficacy of QBETS and detail its ability to automatically take into account job resource characteristics to improve prediction bounds using currently operating large-scale batch systems, and from archival logs for systems that are no longer in operation.
- We describe an implementation of QBETS that provides an on-line batch queue job delay prediction service to high performance computing users and how we have made available a number of programmatic interfaces to the system such that others may trivially integrate QBETS into their own projects.

This ability to make predictions for individual jobs distinguishes our work from other previous efforts. An extensive body of research [7, 11, 12, 18, 19, 20, 23, 60] investigates the statistical properties of offered job workload for various HPC systems. In most of these efforts, the goal is to formulate a *model* of workload and/or scheduling policy and then to derive the resulting statistical properties associated

with queuing delay through simulation. Our approach focuses strictly on the problem of *forecasting* future delay bounds; we do not claim to offer an explanatory, or even a descriptive, model of user, job, or system behavior. However, perhaps because of our narrower focus, our work is able to achieve predictions that are, in a very specific and quantifiable sense, more accurate and more meaningful than those reported in the previous literature.

In Section 4.2, we present a detailed description of QBETS . Section 4.3 discusses our predictor performance experiment, evaluation procedure and the specific results we have achieved. Finally, in Section 4.4 we recap and discuss relevant applications of the methodology.

4.2 Methodology

In this section, we describe our approach to the four related problems that we must solve to implement an effective predictor: quantile estimation[1], change-point detection, job clustering, and machine availability inference. The general approach we advocate is first to determine if the machine of interest is in a state where jobs are being serviced, next to cluster the observed job submission history according to jobs having similar quantitative characteristics (e.g. requested node count, requested maximum execution time, or requested node-hours), then to

[1]We use the term "quantile" instead of the term "percentile" throughout this dissertation.

identify the most recent region of stationarity in each cluster (treated as a time series), and finally to estimate a specific quantile from that region to use as a statistical bound on the time a specific job will wait in queue.

4.2.1 Quantile Prediction

Our goal is to determine an upper bound on a specific quantile at a fixed level of confidence, for a given population whose distribution is unknown. If the quantile were known with certainty, and the population were the one from which a given job's queue delay were to be drawn, this quantile would serve as a statistical bound on the job's waiting time. For example, the 0.95 quantile for the population will be greater than or equal to the delay experienced by all but 5% of the jobs. Colloquially, it can be said that the job has a "95% chance" of experiencing a delay that is less than the 0.95 quantile. We assume that the quantile of interest (0.95, 0.99, 0.50, etc.) is supplied to the method as a parameter by the site administrator depending on how conservative she believes the estimates need to be for a given user community.

However, since the quantiles cannot be known exactly and must be estimated, we use an upper confidence bound *on the quantile* that, in turn, serves as a conservative bound on the amount of delay that will be experienced by a job. To be precise, to say that a method produces an upper 95% confidence bound on a given

quantile implies that the bound produced by this method will, over the long run, overestimate the true quantile 95% of the time. The degree of conservatism we assume is also supplied to the method as a confidence level. In practice, we find that while administrators do have opinions about what quantile to estimate, the confidence level for the upper bound is less meaningful to them. As a result, we typically recommend estimating what ever quantile is desired by the upper 95% confidence bound on that quantile.

Here, we examine the performance of four quantile prediction techniques. The first three are somewhat traditional techniques, each based on fitting a statistical distribution to historical data and using the distribution quantile of interest as the predictor for the next observation. We rely on MLE model fitting of three distributions; log-normal, log-uniform, and Weibull. We note that for the log-uniform and Weibull method, there is no straight-forward way to place confidence bounds on population quantiles and thus we use the model quantile as the predictor. For the log-normal and binomial method predictors, we use the upper 95% confidence bound, but note that even when we use tighter confidence intervals, the resulting predictions are not significantly impacted. The fourth approach is a novel, non-parametric method which makes inference directly from the data, instead of assuming some pre-defined underlying distribution. Here we describe our novel method, which we term the *Binomial Method*, beginning with the following simple

observation: If X is a random variable, and X_q is the q quantile of the distribution of X, then a single observation x from X will be greater than X_q with probability $(1-q)$. (For our application, if we regard the wait time, in seconds, of a particular job submitted to a queue as a random variable X, the probability that it will wait for less than $X_{.95}$ seconds is exactly .95.)

Thus (provisionally under the typical assumptions of independence and identical distribution) we can regard all of the observations as a sequence of independent Bernoulli trials with probability of success equal to q, where an observation is regarded as a "success" if it is less than X_q. If there are n observations, the probability of exactly k "successes" is described by a Binomial distribution with parameters q and n. Therefore, the probability that more than k observations are greater than X_q is equal to

$$1 - \sum_{j=0}^{k} \binom{n}{j} \cdot (1-q)^j \cdot q^{n-j} \tag{4.1}$$

Now, if we find the smallest value of k for which Equation 4.1 is larger than some specified confidence level C, then we can assert that we are confident at level C that the kth value in a sorted set of n observations will be greater than or equal to the X_q quantile of the underlying population – in other words, the kth sorted value provides an *upper level-C confidence bound* for X_q.

Clearly, as a practical matter, neither the assumption of independence nor that of identical distribution (stationarity as a time series) holds true for observed sequences of job wait times from the real systems, and these failures present distinct potential difficulties for our method.

Let us first address the issue of independence, assuming for the moment that our series is stationary but that there may be some autocorrelation structure in the data. We hypothesize that the time-series process associated to our data is *ergodic*, which roughly amounts to saying that all the salient sample statistics asymptotically approach the corresponding population parameters. Ergodicity is a typical and standard assumption for real-world data sets; *cf., e.g.,*[27]. Under this hypothesis, a given sample-based method of inference will, *in the long run,* provide accurate confidence bounds.

Although our method is not invalidated by dependence, a separate issue from the *validity* of our method is that exploiting any autocorrelation structure in the time series should, *in principle*, produce more accurate predictions than a static binomial method which ignores these effects. Indeed, most time-series analysis and modeling techniques are primarily focused on using dependence between measurements to improve forecasting [2]. For the present application, however, there are a number of obfuscating factors that foil typical time-series methods. First of all, for a given job entering a queue, there are typically several jobs in the queue, so that

the most recent available wait-time measurement is for several time-lags ahead. The correlation between the most recent measurement at the time a job enters the queue and that job's eventual wait time is typically modest, around 0.1, and does not reliably contribute to the accuracy of wait-time predictions. Another issue is the complexity of the underlying distribution of wait times: They typically have more weight in their tails than exponential distributions, and many queues exhibit bimodal or multimodal tendencies as well. All of this makes any linear analysis of data relationships (which is the basis of the "classical" time-series approach) very difficult. Thus while the data is not independent, it is also not amenable to standard time-series approaches for exploiting correlation.

4.2.2 History Trimming

Unlike the issue of independence and correlation, the issue of non-stationarity *does* place limitations on the applicability of quantile prediction methods. Clearly, for example, they will fail in the face of data with a "trend," say, a mean value that increases linearly with time. On the other hand, insisting that the data be stationary is too restrictive to be realistic: Large compute centers change their scheduling policies to meet new demands, new user communities migrate to or from a particular machine, *etc.* It seems to be generally true across the spectrum of traces we have examined that wait-time data is typically stationary for a

relatively long period and then undergoes a "change-point" into another stationary regime with different population characteristics. We thus use the Binomial Method as a prediction method for data which are stationary for periods and for which the underlying distribution changes suddenly and relatively infrequently; we next discuss the problem of detecting change-points in this setting.

Given an independent sequence of data from a random variable X, we deem that the occurrence of three values in a row above $X_{.95}$ constitutes a "rare event" and one which should be taken to signify a change-point. Why three in a row? To borrow a well-known expression from Tukey, two is not enough and four is too many; this comes from consideration of "Type I" error. Under the hypothesis of identical distribution, a string of two consecutive high or low values occurs every 400 values in a time series, which is an unacceptable frequency for false positives. Three in a row will occur every 8000 values; this strikes a balance between sensitivity to a change in the underlying distribution of the population and certainty that a change is not being falsely reported.

Now, suppose that the data, regarded as a time series, exhibits some autocorrelation structure. If the lag-1 autocorrelation is fairly strong, three or even five measurements in a row above the .95 quantile might not be such a rare occurrence, since, for example, one unusually high value makes it more likely that the next value will also be high. In order to determine the number of consecutive high

values (top 5% of the population) that constitute a "rare event" approximately in line with the criterion spelled out for independent sequences, we conducted a Monte Carlo simulation with various levels of lag-1 autocorrelation in $AR(1)$ time series [27], observed the frequencies of occurrences of consecutive high and low values, and generated a lookup table for rare-event thresholds. Thus, to determine if a change-point has occurred, we compute the autocorrelation of the most recent history, look up the maximum number of "rare" events that should normally occur with this level of autocorrelation, and determine whether we have surpassed this number. If so, our method assumes the underlying system has changed, and that the relevant history must be trimmed as much as possible to maximize the possibility that this history corresponds to a region of stationarity. Note that indiscriminate history-trimming will not allow our method to function properly, since the resulting small sample sizes will generate unnecessarily conservative confidence bounds.

The minimum useful history length depends on the quantile being estimated and the level of confidence specified for the estimate. For example, it follows from Equation 4.1 above that in order to produce an upper 95% confidence bound for the .95 quantile, the minimum history size that can be used is 59. (This reflects the fact that $.95^{59} < .05$, while $.95^{58} > .05$.)

4.2.3 Job Clustering

According to our observations and to anecdotal evidence provided by users and site administrators, there are differences among the wait times various jobs might expect to experience in the same queue, based purely on characteristics of the jobs such as the amount of time and the number of nodes requested. This is certainly easy to believe on an intuitive level; for example, if a particular queue employs backfilling [40], it is more likely that a shorter-running job requesting a smaller number of nodes will be processed during a time when the machine is being "drained." Thus, for a given job, we might hope to make a better prediction for its wait time if we took its characteristics into account rather than making one uniform prediction which ignores these characteristics.

On the other hand, the same difficulties arise in trying to produce regression models [60] as we encountered in the problem of trying to use autoregressive methods: In particular, the data are typically multimodal and do not admit the use of simple quantile prediction models. We therefore explore the idea of *clustering* the data into groups having similar attributes, so that we can use our parametric and non-parametric predictors on each cluster separately.

In fact, in [5], based on advice we received from several expert site administrators for currently operating systems, we employed a rather arbitrary partitioning of jobs in each queue by processor count, running separate predictors within each

partition, which resulted in substantially better predictions. However, it would clearly be desirable to find a partition which is in some (statistical) sense "optimal" rather than relying on such arbitrary methods; for our purposes, it is also desirable to find a partitioning method that can be machine-learned and is therefore applicable across different queues with different policies and user characteristics without direct administrator intervention or tuning. Moreover, as a diagnostic tool, it would be advantageous to be able to compare the machine-determined clustering with that determined by site administrators to illuminate the effects of administrator-imposed scheduling policies. In this section, we describe our approach to this problem, which falls under the rubric of *model-based clustering* [35, 56, 71].

4.2.4 Model-Based Clustering

The problem of partitioning a heterogeneous data set into clusters is fairly old and well studied [35, 43, 56, 71]. The simplest and most common clustering problems involve using the values of the data, relative to some notion of distance. Often, one postulates that the distribution within each cluster is Gaussian, and the clusters are formed using some well-known method, such as the so-called k-means algorithm [43] or one of various "hierarchical" or "partitional" methods [56, 71]. If the number of clusters is also unknown, a model-selection criterion such as

BIC [57], which we will discuss further below, is often used to balance goodness of fit with model complexity.

In fact, it is tempting, if for no other reason than that of simplicity, to form our clusters in this way, according to how they naturally group in terms of one or more job attributes. Note, however, that this method of clustering in no way takes into account the wait times experienced by jobs, which is ultimately the variable of interest; it is by no means clear that a clustering of jobs by how their requested wait times group will result in clusters whose wait-time distributions are relatively homogeneous. For example, it is possible that a subset of the requested job execution times form a nice Gaussian cluster between 8 and 12 minutes, but that due to some combination of administrative policy, backfilling, and various "random" characteristics of the system as a whole, jobs requesting less than 10 minutes experience substantially different wait times than those requesting more than 10 minutes, so this cluster is actually meaningless in terms of predicting wait times.

In our case, then, the situation is somewhat more complicated than ordinary clustering: We wish to cluster the data according to some characteristics which are *observable at the time the job is submitted* (explanatory variables), but using the actual wait times (response variable) as the basis for clustering. That is, we wish to use observed wait times to cluster jobs, but then to determine how each

cluster is characterized by quantitative attributes that are available when each job is submitted so that an arriving job can be categorized before it begins to wait. In the discussion that follows, we will use the *requested execution time* (used to implement backfilling) as the explanatory characteristic, but this is only for the sake of ease of exposition.

The idea behind our method runs as follows: We postulate that the set of requested times can be partitioned into k clusters $C_1, \ldots, C_k$, which take the form of intervals on the positive time axis, such that within each C_j the wait times are governed by an exponential distribution with unspecified parameter λ_j.

The choice of exponential distributions is something of an oversimplification – in fact a Weibull, log-normal or hyperexponential would probably be a more accurate choice – but the fact that the clusters are relatively homogeneous makes the exponential model accurate enough with relatively little computational expense; moreover, in practice, exponentials are more than discerning enough to produce an adequate number of clusters. As a check, we generated an artificial trace using different log-normally distributed wait times corresponding to the intervals of requested times $[1, 100]$, $[101, 200]$, $[201, 300]$, $[301, 400]$, and $[401, 500]$ and fed this data to our clustering method. It recovered the following clusters for the data: $[1, 39]$, $[40, 40]$, $[41, 100]$, $[101, 197]$, $[198, 300]$, $[301, 398]$, $[399, 492]$, $[493, 493]$, $[494, 500]$. Since our method always clusters the ends together to ensure

that these clusters contain at least 59 elements, the exponential clustering method recovers the original clusters almost exactly.

We assume that the appropriate clustering is into connected intervals along the time axis; this provides an intuitive model for the eventual users of our predictions. Given a desired value for the number k of clusters, then, we use a modified form of *hierarchical clustering*. According to this method, we start with each unique value for the requested time in its own cluster. We then merge the two adjacent (in the sense of adjacency on the time axis) clusters that give the largest value of the *log-likelihood function* $\log L$, calculated jointly across the clusters, according to the maximum-likelihood estimators for the exponential parameters λ_j, which are given by $\frac{\#(C_j)}{\sum_{x \in C_j} x}$. This process continues until the number of clusters is equal to k. Note that this is a well-accepted method for clustering [43, 56, 71]; however, it does not guarantee that the resulting clustering will maximize the log-likelihood over all possible choices of k clusters, even if we assume that the clusters are all intervals. This latter problem is prohibitively expensive computationally for an on-line, real-time application, even for moderately large data sets, and we are therefore forced to use some restricted method.

Each arriving job can then be categorized by identifying the cluster whose minimum and maximum requested time straddle the job's requested time.

Continuing, the question of which value of k to use is a problem in *model selection*, which recognizes the balance between modeling data accurately and model simplicity. The most generally accepted model-selection criterion is the *Bayes Information Criterion* (BIC) [57], the form of which is

$$\mathrm{BIC}(\theta) = \log L(\theta) - \frac{p}{2} \log n,$$

where θ stands for the (vector of) free parameters in the model, L is the joint likelihood function across the whole data set, calculated using the MLE for θ, p is the dimensionality of θ ($2k-1$ in our case: the $k-1$ break points on the time axis to define our clusters, and the k values for the λ_j, all of which are scalars), and n is the total sample size. The first term in the BIC formula should be seen as a measure of goodness of fit, while the second term is a "penalty" for model complexity (*i.e.* one with a large number of parameters). It is always true that for a less restricted model (in our case, one allowing a larger number of clusters), the $\log L$ term will be larger, so the penalty function is critical to avoid over-parameterizing. Maximizing the BIC expression over a set of proposed models has good theoretical properties and generally produces good results in practice. Thus, our clustering strategy is to specify a range of acceptable k-values; perform the hierarchical clustering described above for each of these values of k; and then

calculate the BIC expression for each resulting clustering and choose the one for which BIC is greatest.

4.2.5 Availability Inference

Curiously, it is common for a batch queuing system to continue to accept jobs even when some form of failure has disabled those jobs from being eligible for execution on a set of computation nodes. We know of no automatic detector for this condition that is part of the production batch-scheduling systems used by the machines in our study. Moreover, based on our discussion of this issue with various site administrators, one common solution to this problem seems to be to rely on the users to call when they observe that jobs are no longer being released for execution (even though they can still be queued) and enquire as to whether there is a "problem.". If a ubiquitous service for notification of machine unavailability becomes common, QBETS can trivially be augmented to use such a system. In the meantime, we have found an elegent method to infer machine failures directly from the job waittime data.

To avoid incorporating jobs with artificially lengthened queue delays (due to machine downtime) in the history used for forecasting, QBETS attempts to infer when the computational part of the machine may be down so that these delays can be filtered. Notice that the combination of Binomial-based quantile estimation and

history trimming (sans clustering) provides a relatively general non-parametric method for estimating bounds in time series. QBETS uses this generality in two ways.

First, it counts the number of jobs that have arrived between the points in time when the scheduler releases jobs for execution. As each count is generated, it is incorporated into a time series from which the upper 0.95 quantile (with 95% confidence) is estimated using a Binomial estimator with history trimming. When a count exceeds this upper bound, the QBETS predictor declares the machine to be potentially down until the scheduler releases another job for execution. This functionality is intended to mimic user behavior in which a queue that has been observed to grow "too long" indicates that the computational nodes may be unavailable.

QBETS also maintains a second upper 0.95 quantile predictor to forecast the bounds on the delay between job releases by the scheduler, again using a trimming Binomial estimator. If the time between when jobs are released exceeds the prediction of the bounds, the machine is also marked down until the next job is released. This detector is intended to reflect a user's determination that it as been "too long" since a job was released for execution.

When QBETS temporarily marks a machine as "down", jobs submitted during the down periods are not forecast. Instead, the user is given a signal that can be

interpreted to mean "it is possible that the machine is down at this moment so no prediction is available." Since there is no ground truth as to when the machines in this study were actually down (no failure detector were or are available) it is impossible to know the extent to which this method generates false positive predictions. In general, however, the number of jobs for which "no prediction" would have been returned is a small fraction (usually less than 1%) of the total job submission count.

4.3 Experiments and Results

In this section, we describe our method for evaluating the performance of our chosen batch-queue wait-time prediction system, and we then detail a set of simulation experiments that take as input traces of job submission logs gathered at various supercomputing centers. We describe the details of the simulations and then report the prediction performance that users *would have* seen had the tested system been available at the time each job in each trace was submitted.

We investigate the problem in terms of estimating an upper bound on the 0.95 quantile of queuing delay; however, our approach can be similarly formulated to produce lower confidence bounds, or two-sided confidence intervals, at any desired level of confidence. It can also be used, of course, for any population quantile.

For example, while we have focused on the relative certainty provided by the .95 quantile, our method also effectively produces confidence bounds for the median (*i.e.*, the point of "50-50" probability). We note that the quantiles at the tail of the distribution corresponding to rarely occurring but large values are more variable, hence more difficult to estimate, than those nearer the center of the distribution. Thus, for typical batch-queue data, which is right-skewed with a substantial tail, the upper quantiles provide the greatest challenge for a prediction method. By focusing on an upper bound for the .95 quantile, we are testing the limits of what can be predicted for queue delay.

Note also that our assertion of retroactive prediction correctness and accuracy assumes that users would not have changed the characteristics of the jobs they submitted in response to the availability of the quantile predictions we generate. Moreover, the on-line prototype we have developed, while operational, is in use by only a few users (in fact, we ourselves used QBETS to select which site to execute many of the simulations that generated the results reported here), making it difficult to analyze whether, and how, predictions affect workload characteristics. However, unless such feedback induces chaotic behavior, our approach is likely to continue to make correct and accurate predictions under the new conditions. We do plan to monitor the workloads experienced by various sites after the system is

deployed for general use at various large-scale sites and report on the effects as part of our future work.

4.3.1 Simulation

Our simulator takes as input a file containing historical batch-queue job wait times from a variety of machine/queue combinations and parameters directing the behavior of our models. For each machine/queue for which we have historical information, we were able to create parsed data files each of which contains one job entry per line comprising the UNIX time stamp when the job was submitted, the duration of time the job stayed in the queue before executing, the amount of requested execution time, and the node count.

The steady-state operation of the simulation reads in a line from the data file, makes a prediction (using one of the four prediction methodologies covered in Section 4.2) and stores the job in a "pending queue". The simulation then reads the next job arrival from the input file and, before making a prediction, potentially performs a number of tasks.

First, the simulator checks whether any jobs that had been previously queued have exited the queue since the last job arrived, in which case each such job is simply added to a growing list of historical job wait times stored in memory. Although the waiting time for the new job is carried in the trace, the predictor is

not entitled to "see" the waiting time in the history until it stops waiting in queue and is released for execution. When the historical record changes, the predictor is given the new record so that it can update its internal state, if necessary.

After the queue has been updated, the current prediction value is used to make a prediction for the new job entering the queue, the simulation determines whether the predicted time for that job is greater than or equal to the actual time the job will spend in the pending queue (success), or the predicted time was less than the actual job wait time (failure). The success or failure is recorded, and the job is placed on the pending queue. Note that in a "live" setting this success or failure could only be determined after the job completed its waiting period.

In our first set of experiments, we use only the above simulator features to make predictions for each of the jobs in our traces, varying the predictor used (binomial method, log-normal, log-uniform, and Weibull). For our second set of experiments, we add history trimming, automatic job clustering, and availability inference, as described in Section 4.2, in the following ways.

When a job arrives, the predictor makes a prediction using its current historical window as before and in addition updates the availability inference engine with the current state of the queue, which potentially changes the state of the machine to 'unavailable'. When a job in the pending queue moves into the historical window, it is passed to the predictor, which may then trim the history as

previously described. Every time a pre-determined number (1000 in our study) of simulated jobs are processed, automatic clustering is performed on the entire job history.

The code implementing the simulator is modularized so that any individual component of the system (predictor, history trimming system, clustering algorithm, availability inference algorithm) can be toggled on/off or replaced at runtime. In addition, the nature of the prediction employed methodologies allow the simulator to provide an "on-line" service; meaning it can be executed in a mode where it waits in an idle state until a new job datum arrives, at which point it will update its history and refresh its predictor.

4.3.2 Correct and Accurate Predictions

We define a *correct* prediction to be one that is greater than or equal to a job's eventual queuing delay, and a *correct predictor* to be one for which the total fraction of correct predictions is greater than or equal to the success probability specified by the target quantile. For example, a correct predictor of the 0.95 quantile generates correct predictions for at least 95% of the jobs that are submitted.

Notice that it is trivial to specify a correct predictor under this definition. For example, to achieve a correct prediction percentage of 95%, a predictor could return an extremely large prediction (*e.g.,* a predicted delay of several years) for

19 of every 20 jobs, and a prediction of 0 for the 20th. To distinguish among correct predictors, we compare their *accuracy* in terms of the error they generate, where error is some measure of the difference between predicted value and the value it predicts.

In this work, we will use Root Mean Square (RMS) error for the over-predictions as a measure of accuracy for correct predictors. We consider only over-prediction error, as we believe that the error generated for the percentage of jobs that are incorrectly predicted is relatively unimportant to the user. For example, among predictors that are 95% correct, it is our contention that users would prefer one that achieves lower over-prediction error for the 95% of the jobs it predicts correctly over one that achieves a lower error rate on the 5% that are incorrectly predicted at the expense of greater overall error in the correct predictions.

Note that one cannot compare predictors strictly in terms of their error without taking into consideration their correctness. For example, a predictor that estimates the mean of each stationary region will generate a lower RMS than one that estimates the 0.95 quantile, but the mean predictor will not provide the user with a meaningful delay bound (*i.e.,* one having a probability value attached to it). Thus, for a given job workload, we only compare predictor accuracy among those predictors that are correct.

Note also that, while RMS error is used widely as a measure of accuracy for predictions of expected values (*e.g.* in time series), its meaning is less clear in the context of quantile prediction. In this paper, we are focusing on estimating a time value which is greater than the wait time of a specific job with probability .95. Therefore, if the distribution of wait times is highly right-skewed, a predictor may be working quite well and still have a very high RMS error. Thus, the actual *value* of the RMS error is not particularly meaningful; however, it is still useful as a means of *comparison*: For a particular set of jobs, if one correct prediction method has a lower RMS than another, then the first method, at least by this measure, produces tighter, less conservative upper bounds than the second.

4.3.3 Experiments

We perform two experiments in order to show the effectiveness of our prediction methods. The first experiment compares the correctness and accuracy of four different predictors for all data sets without the use of history trimming, job clustering or availability inference features. The results of this experiment are shown in Table 4.1. From the table, we first note that while each of the predictors is correct for some subset of the traces, the only predictor that is correct for all traces is the one based on the log-uniform distribution. Thus it might appear that the log-uniform-based method is the obvious winner for batch-queue prediction;

however, upon closer inspection it becomes clear that the only reason this method is getting 95 percent or more of the predictions correct for any given trace is due to its extremely conservative individual predictions. This fact is reflected in the extremely low RMS ratios for the log-uniform method shown in Table 4.1 under **Accuracy**, which clearly indicates that the distance between the log-uniform predictions and the actual values is much greater than, say, the distance between the binomial method predictions and actual values for the same set of jobs. Note that in the table, bold values indicate that the shown method also was correct for that machine/queue/predictor tuple. The over-conservativeness of the log-uniform predictions is also borne out by the fact that, in general, its fraction of correct predictions is well above the target value of .95.

From the first set of experiments, we learned that there is no method that is both more correct and more accurate than the others. Our second experiment uses a combination of all of the features we have developed to improve both the correctness and the accuracy of each of the techniques. In Table 4.2, we show the results of the QBETS system on the same traces, varying only the predictor used during the simulation. Again, values in bold indicate machine/queue/predictor tuples which were correct. From these results, we can begin to see that the binomial method clearly stands apart from the rest in terms of both correctness and accuracy. Out of 25 traces, the binomial method was correct 22 times, which

is more often than all others except for the log-uniform. Further, note that out of the 21 traces for which both the binomial and log-uniform methods were correct, the binomial was more accurate for every one of them. Additionally, overall, the binomial method was both correct and more accurate than any of the other predictors in 15 out of 25 traces; this number far exceeds the performance of any other predictor (log-normal 2/25, log-uniform 2/25, Weibull 5/25).

4.3.4 Correctness Analysis

Table 4.2 shows that when we use QBETS with the binomial-method predictor, we are able to predict bounds correctly for 95% or more individual job wait times for almost all of our traces. In this section, we explore the reasons for the effectiveness of QBETS and suggest that, for these reasons, the non-parametric approach should perform well when applied to other traces in the future.

In previous work [4], we showed that using history trimming is essential to ensure that a predictor not suffer from an inability to adjust to drastic infrequent increases in overall job queue wait times. In Figure 4.1, we can see the effect such drastic regime shifts have on a predictor without history trimming, and observe how trimming positively effects correctness on an example trace, the CNSI Dell cluster default queue (cnsidell/ALL). On the y-axis we show delay measured in seconds. Along the x-axis are Unix time stamps. The relatively straight line

of values near the bottom of the graph depicts .95 quantile predictions made by the binomial method, but without QBETS enhancements, during a short time period. We can see that although a large number of observations lie above these predictions in the right half of the graph, there are enough relatively low values in the history that the inferred .95 quantile rises only very slowly. The other set of predictions, represented on the graph by a number of near-horizontal short segments, were made by the binomial method with QBETS over the same time period, is able to react to the shift toward longer wait times and is therefore able to produce more correct predictions. In general, this adaptivity greatly improves a predictor's ability to achieve its desired correctness, because such shifts are common in almost all of our traces. We note that while history trimming is an effective enhancement for all of the predictors, it works especially well with the binomial predictor; we posit that this is due to the fact that the binomial predictor is set up to make accurate inferences about quantiles, so that it is able to find changepoints in those quantiles reliably. In essence, the accuracy of the method (*Cf.* Section 4.3.5) feeds its correctness.

Although QBETS allows the predictor to react to drastic wait-time shifts, there are still traces for which it fails to meet the target percentage of correct predictions. In the cases where QBETS fails, we observe that in general, the reason is due to frequent drastic upward trends in wait times, which appear as

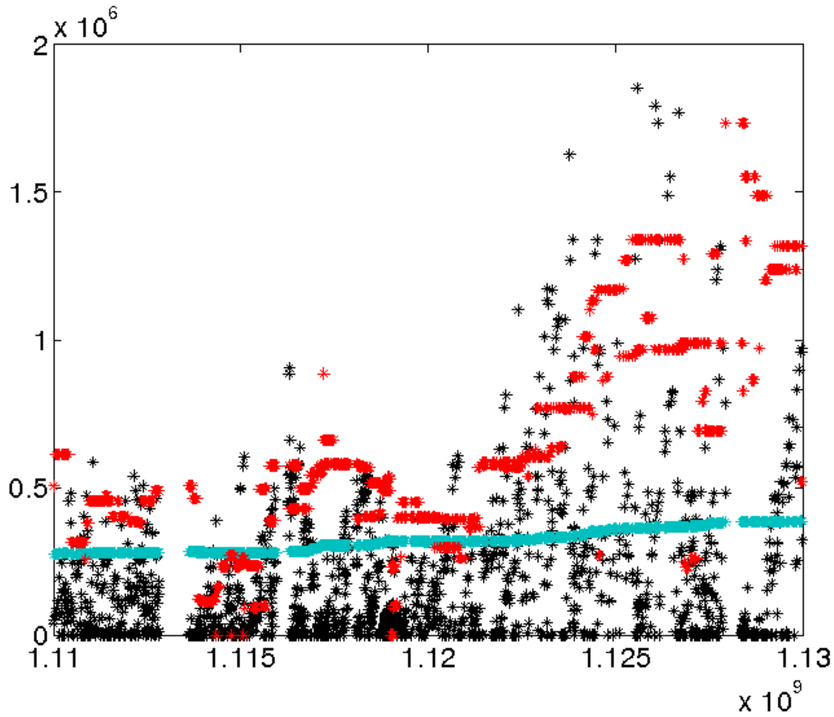

Figure 4.1: Job queue delay times and predictions made with and without QBETS on the CNSI Dell cluster. Dark features (black) indicate actual job wait times, the medium shaded (cyan on color displays) linear features depict predictions made without QBETS , and the light colored features (red) depict predictions made with QBETS .

'spikes' in the trace graphs. Figure 4.2 shows such spikes in the middle of the Dante default queue trace. As we can see from this graph, if a large number of jobs is queued in a relatively short amount of time, and all of them experience wait times that are greater than the current quantile prediction, our method will fail to correctly make predictions for most of them, due to the fact that a wait time is not added to the predictor's available history until it comes out of the queue. Although the availability inference method attempts to discover these degenerate

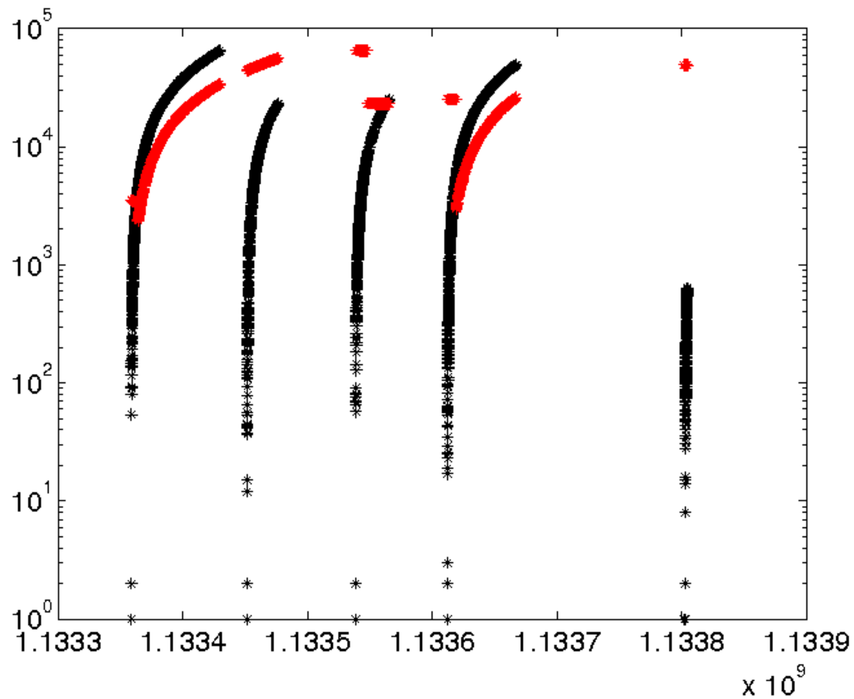

Figure 4.2: Actual queue delay times and QBETS binomial method predictions illustrating how frequent, drastic linear delay increases on the Dante cluster cause the method to fail. Dark features (black) show actual observed job wait times, while the light features (red on color displays) depict QBETS predictions.

data cases, it cannot discover them all. In the traces for which QBETS with binomial predictor is unable to succeed, such as the dante default queue trace shown here, there are many spikes that the availability inference method does not eliminate; their negative impact on the overall correctness measure outweighs the number of jobs the method does correctly capture.

While one might be tempted to use an extremely conservative method in order to combat this eventuality, this strategy may require such extreme measures as to

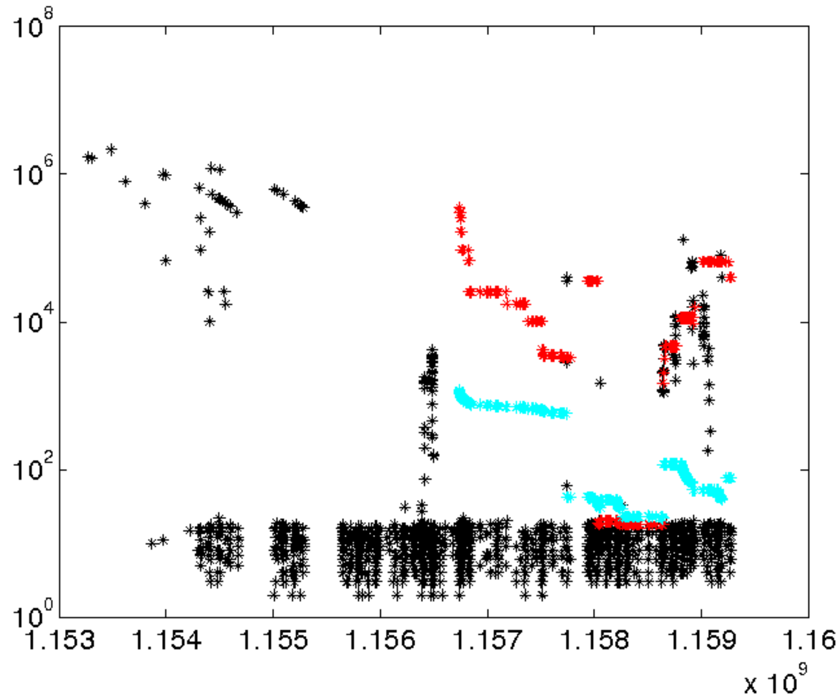

Figure 4.3: Trace from the Tsubame machine, Gaussian submission queue indicating large difference between log-normal and binomial method QBETS predictions after the training period. Dark features (black) show actual observed job wait times, medium shaded features (cyan on color displays) depict less conservative predictions using the log-normal, and light features (red) show predictions when the binomial method was used.

make the method unreasonable for non-degenerate cases. We note that even the log-uniform method, which is the most conservative method we evaluate, fails to be correct in the face of the dante default queue trace.

4.3.5 Accuracy Analysis

In terms of accuracy, the results presented above support two assertions. First, QBETS is the most accurate of the methods we have tested. Second, the non-parametric binomial quantile estimator is more effective than the corresponding parametric approaches. That is, when the change-point detection, clustering, and machine downtime detection features of QBETS are omitted, and we are simply applying the binomial prediction method to all jobs using the entire history, the binomial method still provides more accurate over-predictions than the other methods.

This greater accuracy, we believe, is because the binomial technique estimates directly only a specific quantile and not the entire distribution. In contrast, parametric approaches using MLE attempt to "fit" the data to all quantiles and in so doing may not estimate the specific quantile of interest as accurately. In particular a log-normal or Weibull model such as we have chosen to evaluate in this experiment (and typically used for such highly right-skewed data as in our traces) suffers from the fact that quantiles out in the tail of the distribution are very sensitive to the estimated population parameters. For the same reason, using a estimation technique such as MLE, the estimated parameters are sensitive to a few very high values in the data set. Thus an estimated quantile for such a distribution is highly dependent on the model's ability, typically based on a small

number of high values in the data, to fill in its right tail. In practice, the end result of this phenomenon is usually that the quantile estimates produced by these parametric models are much more conservative than the ones that can be made using the binomial method, which does not need to take into account the relationship between high and (irrelevant for our purposes) low values in the way that curve fitting does.

One fundamental reason for the superior accuracy of predictions generated using QBETS stems from the automatic job-clustering feature, which allows the predictor to only consider "like jobs" when making its prediction instead of all jobs, which may be only loosely related to the job of interest in terms of experienced wait time. During the experiment, we observed that QBETS automatically grouped jobs into three to five clusters, never choosing only one group for all jobs. Additionally, we observe that not only is QBETS more correct in general, but that QBETS with the binomial method predictor outperforms the other predictors in most of the traces. Again, the reason this is true is due to the fact that in general the binomial method is making more accurate predictions, as we see from Table 4.2 and Table 4.1; this amounts to heightened sensitivity to change-points in the data, thus allowing the history-trimming feature to activate more often than it does for other predictors.

This being said, there are a few cases where the parametric models were in fact **more** accurate than the binomial method. In these cases, most notably the tsubame/guassian and tsubame/high traces, we observe that the primary reason why the log-normal is achieving so much better RMS errors stems from the fact that in those traces, the training period data included a disproportionate number of very large wait times relative to the experimental set. The training set can be seen in Figure 4.3 as the period of observations before any predictions are being made; notice that the binomial method starts out making very conservative predictions based on the large number of high values in the training set, while there are enough low values to bring down the MLE log-normal parameters, making these predictions less conservative. In this case, data for the training period was bimodal, with about 10% of the wait times in an extremely high mode, orders of magnitude higher than the bulk of the wait times in the lower mode. This higher mode, which would have caused the log-normal predictions to be incorrect, disappeared at the end of the training period, leaving the binomial method with an unrepresentative data set to begin with and also rendering the log-normal predictions both correct and accurate. We note two things, however: First, the experimental set was only slightly larger than the training set, so that there was not time to balance the anomalies in the training data, and so may not have been reflective of long-term performance; second, by the middle of the experimental

set, the binomial method predictor was able both to make more accurate predictions than the log-normal predictor for the relatively short wait times and also to maintain correctness when the wait times suddenly became longer again at the end of the trace.

4.4 Conclusions

Space-shared parallel computers use queuing systems for scheduling parallel jobs to processor partitions in such a way that each job runs exclusively on the processors it is given. While such techniques have been optimized for resource utilization, the amount of time individual jobs wait in queue is highly variable and often comprises a substantial portion of the overal-turnaround time of a job.

In this chapter we describe QBETS , which combines history trimming, automatic job clustering, availability inference, and various prediction methodologies to provide a batch queue job wait time prediction system which is shown to perform better than more naive approaches for almost all of the data we have access to. Additionally, we show that QBETS , with the non-parametric binomial method quantile predictor presented in previous work, is both more correct and more accurate than any other tested technique and prediction method.

While QBETS offers users the ability to predict bounds on the amount of time individual jobs will wait in queue, and further to predict the probability of a job meeting a specified deadline, there are other problems that QBETS alone does not address. QBETS predicts an upper bound on job wait time, but does not inform the user when, during that interval, the job is likely to run. For users who require resources or jobs to be available during a specific time interval, the QBETS is not sufficient to satisfy their requirements. In the next chapter, we describe this problem in more detail and present our solution.

Machine	Correctness				Accuracy			
Queue	BM	LogN	LogU	Weib	BM	LogN	LogU	Weib
cnsidell								
ALL	0.92	0.97	0.97	0.81	1.00	**0.21**	**0.48**	2.14
dante								
dque	0.82	0.75	0.96	0.40	1.00	1.28	**0.48**	8.82
datastar								
TGnormal	0.91	0.83	0.98	0.84	1.00	4.16	**0.25**	3.51
express	0.93	0.88	1.00	0.84	1.00	3.16	**0.11**	3.90
high	0.90	0.92	0.97	0.85	1.00	0.74	**0.27**	1.48
normal	0.91	0.91	0.99	0.88	1.00	0.90	**0.17**	1.37
ucteragrid								
dque	0.89	0.88	1.00	0.94	1.00	11.28	**0.00**	12.30
lonestar								
development	0.92	0.92	1.00	0.92	1.00	3.30	**0.00**	4.40
high	0.96	0.98	1.00	0.94	**1.00**	**0.61**	**0.22**	1.54
normal	0.92	0.84	1.00	0.84	1.00	4.00	**0.04**	4.74
serial	0.97	0.95	1.00	0.94	**1.00**	**2.77**	**0.03**	4.54
ncsateragrid								
debug	0.93	0.88	0.99	0.91	1.00	2.02	**0.14**	0.59
dque	0.93	0.89	1.00	0.91	1.00	1.06	**0.06**	0.51
gpfs-wan	0.99	1.00	1.00	0.93	**1.00**	**0.16**	**0.55**	0.66
sdscteragrid								
dque	0.93	0.86	0.98	0.90	1.00	2.44	**0.23**	0.26
tsubame								
B	0.93	0.94	1.00	0.94	1.00	11.38	**0.00**	4.22
default	0.93	0.84	1.00	0.84	1.00	13.16	**0.01**	6.22
gaussian	0.96	0.94	1.00	0.95	**1.00**	137.71	**0.08**	**23.15**
high	1.00	0.97	1.00	0.97	**1.00**	**210.66**	**0.14**	**37.95**
ctc								
ALL	0.94	0.97	1.00	0.92	1.00	**0.48**	**0.04**	0.49
llnl								
ALL	0.96	0.99	1.00	0.94	**1.00**	**0.29**	**0.08**	0.63
sdscblue								
high	0.90	0.90	1.00	0.79	1.00	0.53	**0.15**	1.51
low	0.90	0.99	1.00	0.89	1.00	**0.36**	**0.11**	1.09
normal	0.89	0.94	1.00	0.85	1.00	0.44	**0.09**	1.13
express	0.92	0.90	0.99	0.84	1.00	1.12	**0.17**	2.20

Table 4.1: Correctness and accuracy results of four predictors without QBETS . Under **Correctness**, values >= 0.95 indicate a correct result. Under **Accuracy**, highest RMS error ratio indicates most accurate method.

Machine *Queue*	Correctness				Accuracy			
	BM	LogN	LogU	Weib	BM	LogN	LogU	Weib
cnsidell								
ALL	0.96	0.93	0.93	0.97	**1.00**	0.05	1.55	**0.51**
dante								
dque	0.80	0.69	0.87	0.72	1.00	0.05	0.71	0.40
datastar								
TGnormal	0.97	0.90	0.97	0.96	**1.00**	0.46	**0.85**	**0.70**
express	0.97	0.87	0.99	0.93	**1.00**	0.78	**0.59**	1.28
high	0.96	0.95	0.98	0.95	**1.00**	**0.31**	**0.78**	**0.75**
normal	0.95	0.92	0.97	0.93	**1.00**	0.23	**0.65**	1.00
ucteragrid								
dque	0.96	0.94	1.00	0.96	**1.00**	0.25	**0.19**	**0.78**
lonestar								
development	0.98	0.92	1.00	0.96	**1.00**	2.12	**0.07**	**2.85**
high	0.98	0.95	1.00	0.96	**1.00**	**0.34**	**0.29**	**0.81**
normal	0.96	0.89	0.99	0.94	**1.00**	0.07	**0.50**	0.66
serial	0.97	0.81	1.00	0.92	**1.00**	1.17	**0.21**	0.49
ncsateragrid								
debug	0.96	0.86	0.98	0.91	**1.00**	1.37	**0.55**	2.02
dque	0.93	0.91	0.97	0.93	1.00	0.17	**0.46**	1.06
gpfs-wan	0.92	0.98	1.00	0.93	1.00	**0.38**	**0.66**	0.96
sdscteragrid								
dque	0.96	0.88	0.98	0.93	**1.00**	0.12	**0.89**	0.50
tsubame								
B	0.98	0.91	1.00	0.97	**1.00**	2.45	**0.29**	**1.27**
default	0.97	0.94	1.00	0.96	**1.00**	0.05	**0.18**	**0.73**
gaussian	0.98	0.95	1.00	0.97	**1.00**	**177.20**	**0.08**	**8.37**
high	0.99	0.97	1.00	0.98	**1.00**	**70.46**	**0.17**	**18.76**
ctc								
ALL	0.96	0.93	0.99	0.93	**1.00**	0.48	**0.18**	1.66
llnl								
ALL	0.97	0.95	0.99	0.95	**1.00**	**0.65**	**0.57**	**1.58**
sdscblue								
high	0.96	0.96	0.97	0.94	**1.00**	**0.24**	**0.87**	0.97
low	0.96	0.96	0.99	0.95	**1.00**	**0.26**	**0.38**	**1.08**
normal	0.97	0.95	0.97	0.95	**1.00**	**0.24**	**0.55**	**1.03**
express	0.97	0.91	0.98	0.94	**1.00**	0.17	**0.50**	0.55

Table 4.2: Correctness and accuracy results of four predictors using QBETS . Under **Correctness**, values >= 0.95 indicate a correct result. Under **Accuracy**, highest RMS error ratio indicates most accurate method.

Chapter 5

Virtual Advance Reservations

5.1 Introduction

One approach to solving the planning problems brought about by unpredictable queuing delay is to allow users to make *advanced reservations* [32, 59, 61] for resources. With an advanced reservation system in place, users can attempt to reserve partitions of the machine, each starting at a particular time for a specified duration. In situations where real-world deadlines are critical to success (e.g. for paper deadlines, conference demonstrations, collaborative meetings, etc.) an advanced reservation capability is essential.

However, while most open-source and commercial batch schedulers provide support for user reservations, to date this capability is not offered to the general user population by any of the HPC computing centers of which we are currently aware. While there are a number of reasons why advanced user-settable reser-

vations are not available (*e.g.*, it is not clear what users should be charged for a reservation, what the priorities for making a reservation should be, etc.) the primary concern appears to be the possible loss of machine utilization. Unlike a busy restaurant that can cover the cost of an unused or under-used reservation through higher prices to all customers, the HPC centers pay for their resources almost entirely "up front" and then account for the capital expense as utilization over the lifetime of the machine. Thus lost utilization can be viewed as lost revenue that cannot be recovered. It is currently true that specially privileged users may still make reservations for particularly important and well-justified deadlines, but these reservations are negotiated individually with site administrators beforehand on a case-by-case basis and are not available to the general user community.

In this Chapter, we present a new statistical method that implements advanced reservations probabilistically as an overlay atop existing best-effort (i.e. non-reservable) batch-queue systems in production HPC settings. Our approach builds upon recent work in predicting *bounds* on queuing delay using fast, on-line time-series techniques [48]. We use these results to build a *virtual reservation* capability – **V**irtual **A**dvanced **R**eservations for **Q**ueues (VARQ) – for regular (e.g. non-privileged) users that does not require the cooperation of the target batch scheduler. With VARQ, site administrators are not required to implement a local reservation capability; rather, they see jobs managed by our system as part of the

normal workload. Users experience the cost of a virtual reservation as an additional charge to their accounts (typically funded in units of allowed occupancy time) automatically, without a change to local accounting systems. One drawback of our approach is that the exact cost for a specific reservation is difficult to predict precisely. The system attempts to minimize this cost, however, and it does provide conservative worst-case estimates. Finally, users are able to specify explicitly an acceptable failure probability for each VARQ virtual reservation.

In this Chapter, we detail the implementation of virtual advanced reservations and evaluate its effectiveness empirically using several shared production HPC facilities currently dedicated to science and engineering research. We also analyze the cost, in terms of additional charges to our occupancy allocations, incurred during our experiments. Finally, we use a trace-based, faster-than-real-time simulator to explore the possible effects of virtual advanced reservations should our system become a popular infrastructure component.

In so doing, the work described in this Chapter makes the following contributions.

- We propose a statistical approach to implementing advanced reservations in production science and engineering HPC settings that does not require site administrators to implement hard reservations.

- We analyze the effectiveness of this approach using both a working implementation targeting "live" HPC systems running in production mode and its potential impact using a new trace-based simulation capability.
- We find that virtual advanced reservations are surprisingly effective at the present time on the HPC machines we tested and that their impact is unlikely to affect current HPC operational settings negatively.
- We describe the statistical conditions that must exist at the sites for these results to be general in future, and argue that they are likely to exist for the near and medium term.

These contributions are important and relevant to the parallel computing community because they offer the possibility of providing a specific quality-of-service to users without the need to modify local software and/or management policies. In particular, for grid settings where resource usage is federated and cross-site scheduling is tremendously challenging, we believe virtual reservations prove an important and enabling technology.

The remainder of this chapter is organized as follows. In the next section (Section 5.2, we describe the statistical approaches and methods we have developed to make virtual advanced reservations. Section 5.3 describes the experiments we have performed to evaluate the efficacy and generality of our technique and their

results. Finally we conclude this chapter with a discussion of limitations of our method in Section 5.4.

5.2 Methodology

VARQ implements a reservation by determining when (according to predictions made by QBETS) a job should be submitted to a batch queue so as to ensure it will be running at a particular point in future time. It does not actually reserve the resource, but rather achieves the same goal – the predictably scheduled execution of a user program – that a reservation enables. Because VARQ does not require modification to the local scheduling policies or scheduler submission protocols, it can function as an overlay. VARQ jobs do not appear different from non-VARQ submissions at the batch scheduler.

Figure 5.1 provides an overview of the functional relationships between VARQ, QBETS, and the local batch scheduler. User batch jobs may be submitted to VARQ for execution at a specific point in time in the future (specified by a deadline). VARQ then uses QBETS predictions to determine when the job should be submitted to the batch scheduler queue to ensure it is executing at the deadline. At the same time, non-VARQ jobs are being submitted to the batch scheduler queue. The delays these jobs experience affect QBETS predictions which in turn,

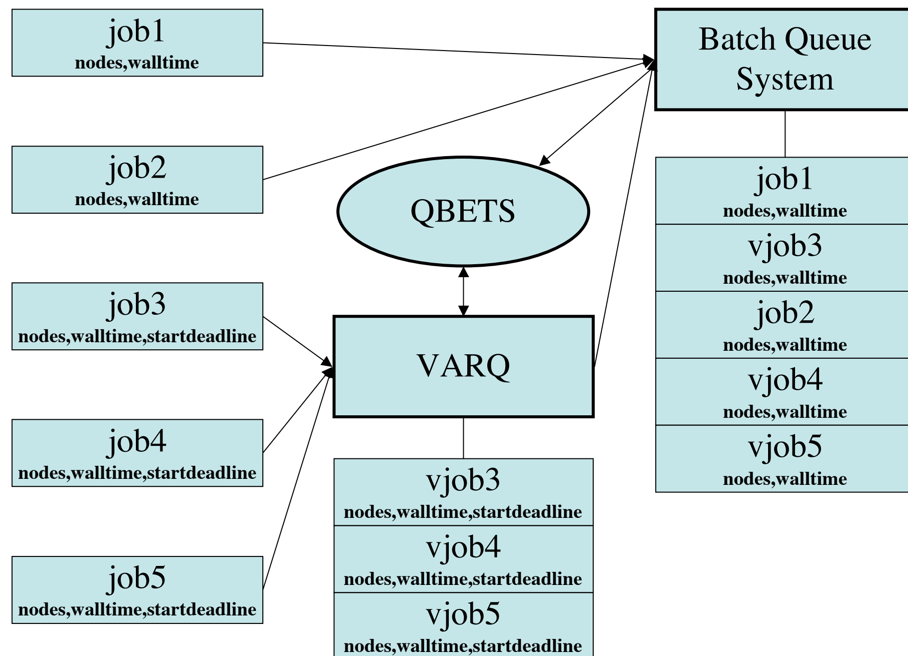

Figure 5.1: Overview of VARQ system interaction with local batch scheduler

affects VARQ decision making. We describe the nature of this interaction more completely in the following subsections.

5.2.1 Virtual Advanced Reservations

Using QBETS, we can estimate the probability, at time T, of a specific job beginning execution by a certain time in the future $T + startDeadline$, but we cannot say when between T and $T + startDeadline$ the job will actually start. Us-

ing the UC TeraGrid example above, we know that, at time T, QBETS reported a 0.89 probability of the specified job starting within ten minutes. However QBETS provides no information about the likelihood of the job starting at any specific time between $T+1$ seconds and $T+599$ seconds. Because of this uncertainly, this probabilistic prediction alone is not sufficient for certain applications which need to reserve a precise time slot in the future when the resources will be available.

One naive way to get around this deficiency is to attempt to submit a job that requests a runtime long enough to encapsulate both the time from T to $T + startDeadline$ and the requested time of the job itself (wallTime). Using such a tactic, we would submit a job which, instead of requesting $wallTime$ seconds of compute time, instead requests $wallTime + startDeadline$ seconds. This technique will guarantee that if the job begins execution between T and $T + startDeadline$, then it will be allowed to execute from $T + startDeadline$ to $T + startDeadline + wallTime$. If the user desires that the job start at $T + startDeadline$ and not before, the job simply needs to "sleep" or spin until time $T + startDeadline$. Recall from Section 5.1 that once the batch scheduler allocates nodes to a job, the job will not be prematurely terminated nor pre-empted. Thus any job is free to simply wait to begin doing useful work, however the accounting system will charge the user's allocation for occupancy starting at the moment the job acquires its nodes. Because this occupancy is by a user job,

and the user's account is charged for it, the center does not need to, and indeed cannot, consider it lost utilization.

Potential Drawbacks

The disadvantages to this approach are twofold. First, when the desired $startDeadline$ is large (and it likely is), then the job could waste a substantial amount of allocation by holding the resources until the user's deadline arrives. Second, again when $startDeadline$ is large, the probability of such a large job making it through the queue is much lower than the job requesting the time actually needed for execution. For example, if $startDeadline = 43200$ and $wallTime = 3600$, then we would be requesting a 46800 second job when we only need 3600 seconds of compute time, 43200 seconds from now. The probability of a 46800 second job making it through the queue by $startDeadline$ is much lower than that of a 3600 second job starting by $startDeadline$, primarily because of the inability of the scheduler to use this job for backfilling.

Bounds Prediction Stability

Our solution to this problem is to find the amount of time to wait before submitting a job so that when we do submit, the job isn't so large as to make the probability of success prohibitively low because of the additional runtime necessary

to cover the possibility that it begins running immediately. This approach is based on the observation that if the percentile estimates do not change, or change slowly, QBETS predictions made in the future will look very much like current predictions. Thus it is possible to predict the bounds on delay if the user were to wait a short time (thereby reducing the extra time needed to cover the delay until the deadline) before submitting a job.

In the process of verifying QBETS (a process that continues), we observed that while the queue delays may fluctuate to a great degree, the time series of percentile predictions corresponding to those delays are relatively stable, often over many days. Figure 5.2 presents an example that compares queue delays observed for jobs submitted to the "normal" queue (the default work queue) on the San Diego Supercomputer Center's Datastar machine and the corresponding QBETS estimates for the upper 95th percentile reported during the month of February of 2007.

In the figure, the x-axis represents the submission time of a job, and the y-axis (using a log scale) describes the delay (so that the figure shows the *time series* of delays). Each point feature represents the delay observed for a single job, and the line feature traces the QBETS estimates. Notice that even though the job delays vary from between 10 and $110,000$ seconds, the percentile estimate is quite stable by comparison.

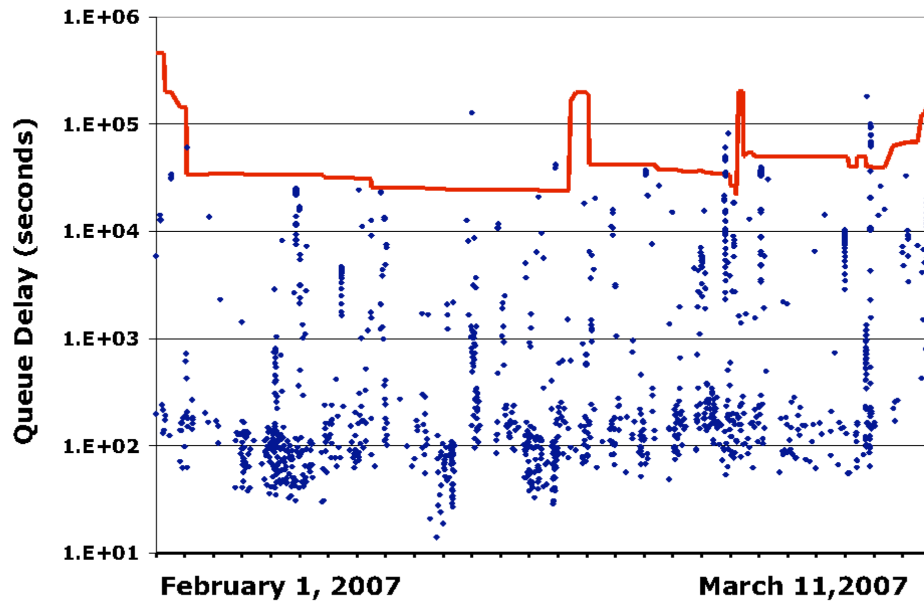

Figure 5.2: Queue delay measurements and QBETS 95th percentile predictions on Datastar for the month of February, 2007.

The clustering feature of QBETS enhanced the stability shown in Figure 5.2 by selecting a relatively homogeneous subset of the wait times. While the full trace shows several orders of magnitude variation, this trace for a single cluster automatically identified by QBETS shows only a range of only about 4 orders of magnitude. The reason for this effect is that the appearance of highly variable delay may be because the scheduler is interleaving jobs of different classes, each of which experiences a different "class" of delay. For example, if large jobs are

experiencing roughly $100,000$ seconds of delay and small jobs are all being serviced in 10s of seconds, an interleaving of the two appears to have more variance than either taken separately.

QBETS, as discussed in Chapter 4, computes a time bound on the delay a specific user job will experience. For this work, we have modified QBETS to invert this functionality so that it returns an integer percentile estimate between 1 and 100 for a specific time bound. By treating these percentiles as coming from a single empirical distribution, QBETS can return the probability that a job corresponding to a specific 5-tuple will begin executing before a specified period of time has elapsed (termed the deadline). This functionality, which is the foundation of VARQ, is expressed here as a function to be used for the remainder of this dissertation:

$$QBETS(m, q, nodes, wallTime, startDeadline) = prob$$

Because the bounds predictions are so stable, it is possible to use the inverted predictor function $QBETS()$ to estimate the probabilities that jobs submitted at successive points in the future (each having a successively shorter requested execution time) will start running at some point before a specific deadline and will be able to continue executing until completion. However, the effect is not

monotonic. Notice that in Figure 5.2 very few jobs waited less than 20 seconds between the time they were submitted and the time they began execution; a greater number waited between 20 and 100 seconds; *etc.* This effect occurs because in the short run, the scheduler attempts to implement a fair policy between jobs of equivalent resource requirements. Thus a VARQ job submitted near the deadline will contend with non-VARQ jobs for immediate initiation thereby, decreasing that job's probability of starting before the deadline. Therefore, as the submission time approaches the deadline, the probability of starting before the deadline tends to *increase*, possibly due to the backfilling, as less additional runtime is necessary to cover the time before the approaching deadline; but it tends to *decrease* due to contention by other submissions and the scheduler's need to enforce fairness among equivalent jobs. One might expect, then, to find a "sweet spot" at which the probability is maximized.

Probability Trajectories

To find the submission time in the future that will most likely meet the deadline or a submission time that corresponds to a user's reservation request, VARQ computes a *probability trajectory* for the user's job by considering the possibility of submitting a given job at successive 30 second intervals from the time the job is given to VARQ until the specified deadline. For each point in time, it decre-

ments the additional runtime $startDeadline$ required by 30 seconds, and estimates the probability of starting before the deadline using $QBETS()$. Specifically, it implements the algorithm described in pseudocode in Figure 5.3, where the algorithm accepts as input the 4-tuple job description, the required time when the resources must be available ($startDeadline$) and the minimum acceptable probability that the reservation request is successful ($reqProb$). Upon completion, the algorithm returns the number of seconds VARQ should wait before submitting the job ($waitT$), and the modified job walltime ($advWallTime$) required to ensure that the reservation can be met with probability $reqProb$.

Figure 5.4 depicts an example VARQ probability trajectory (denoted $probVec$ in the pseudocode) computed in this way. The data comes from a VARQ reservation made at 2:49 PM on March 6th, 2007 on the NCSA TeraGrid machine for the "dque." For this reservation, the user requested 4 nodes for 1 hour of execution time starting at 2:49 AM on March 7th (12 hours into the future). Time of day (given as a Unix timestamp) beginning at 2:49 PM on the left-hand side of the figure is shown along the x-axis. The y-axis shows the return values of $QBETS()$ which is the probability estimate for the job starting before the deadline at 2:49 AM (right-hand side of the graph) as function of when, in the future, it is submitted.

```
INPUT(mach, queue, nodes, wallTime, startDeadline, reqProb)
OUTPUT(waitT, advWallTime)
T = current UNIX timestamp
currT = T
currProb = I = 0

WHILE (currT < startDeadline)
   advWallTime = wallTime + (startDeadline - currT)
   currProb = QBETS(nodes, advWallTime)
   probVec[I] = (currT, currProb)
   I = I + 1
   currT = currT + 30
ENDWHILE

I = LENGTH(probVec)
WHILE (I >= 0)
   (currT, currProb) = probVec[I]
   IF (currProb >= reqProb) THEN
      waitT = currT
      advWallTime = startDeadline - currT
      RETURN(waitT, advWallTime)
   ENDIF
   I = I - 1
ENDWHILE
```

Figure 5.3: Pseudocode describing VARQ determines how long to wait before submitting a VARQ job.

From time T at 2:49 AM until approximately $T+21600$ seconds, the probability of the requisite sized job starting before $T+43200$ steadily drops from slightly below 0.4 to 0.25. This part of the graph illustrates the probability decay that occurs as the eventual submission time and the deadline draw closer together. However, at approximately $T+21600$, we see a drastic increase in probabilistic prediction. This increase shows the effects of the clustering algorithm used by QBETS. At that point in time, the combination of node request and total requested execution

time for the job put it in a different scheduler service class (presumably due to the possibility of backfilling). After this point, again the probabilities steadily approach 0 as the deadline approaches.

The probability trajectory can be used to identify the point in time when VARQ should submit the job to the machine's queue that corresponds to the most probable success in attaining the reservation (at $T + 21600$ in the figure the QBETS reported a maximum probability of approximately 0.7). VARQ supports this mode of operation, but in choosing the maximum, the user cannot explicitly tradeoff success probability for potentially lost allocation. If the user in this case requested VARQ to submit at the most probable point in time, and the job began running immediately, the user's allocation could be charged a maximum of an additional $4 * (43200 - 21600) = 86400$ node-seconds of allocation in addition to the $3600 * 4 = 14400$ node-seconds required for the job's execution.

Minimizing Lost Allocation

To allow somewhat greater efficiency and flexibility, VARQ also accepts a target success probability from the user and finds the *latest* submission time in the probability trajectory that can meet it as a way of minimizing the additional allocation overhead. For example, if the user specified success probability of 0.5 and the trajectory in Figure 5.4 were used, then VARQ walks backwards in $probVec$

Machine	Processors	Batch Software	Description
datastar	2176	Load Leveler	SDSC IBM PowerPC Production Compute Cluster
ucteragrid	316	Torque/Maui	UC/ANL IBM/Intel Compute Viz Linux TeraGrid Cluster
dante	35	Torque/Maui	RENCI Intel Xeon Research Linux Cluster
cnsidell	256	Torque/Maui	UCSB NanoScience Research Linux Cluster
ncsateragrid	1744	Torque/Maui	NCSA IBM/Intel Compute Linux TeraGrid Cluster
iuteragrid	32	PBS	IU AVIDD Compute Linux Cluster
ornlteragrid	56	Torque	ORNL IA64 Compute Linux Cluster

Table 5.1: HPC machines used in VARQ empirical experiment. Chosen systems represent a realistic set of distributed HPC resources on which users would have simulataneous access.

until it encounters the first timestamp where the probability is equal to or exceeds 0.5. In this case, such a timestamp exists at $T + 40470$ which indicates that we should wait 40470 seconds and then submit a 4 node job requesting 6330 second job $(3600 + (43200 - 40470))$ to guarantee that the job will be running between $T + 43200$ and $T + 43200 + 3600$. The potential allocation overhead (i.e. the maximum possible additional allocation cost) is $4 * (43200 - 40470) = 9480$ node-seconds for the same job requiring 14400 node-seconds of execution time.

Thus the VARQ probability trajectory allows the user to trade estimated success probability for potential allocation overhead. In this example, reducing the desired success probability from 0.7 to 0.5 implies a reduction in potential extra

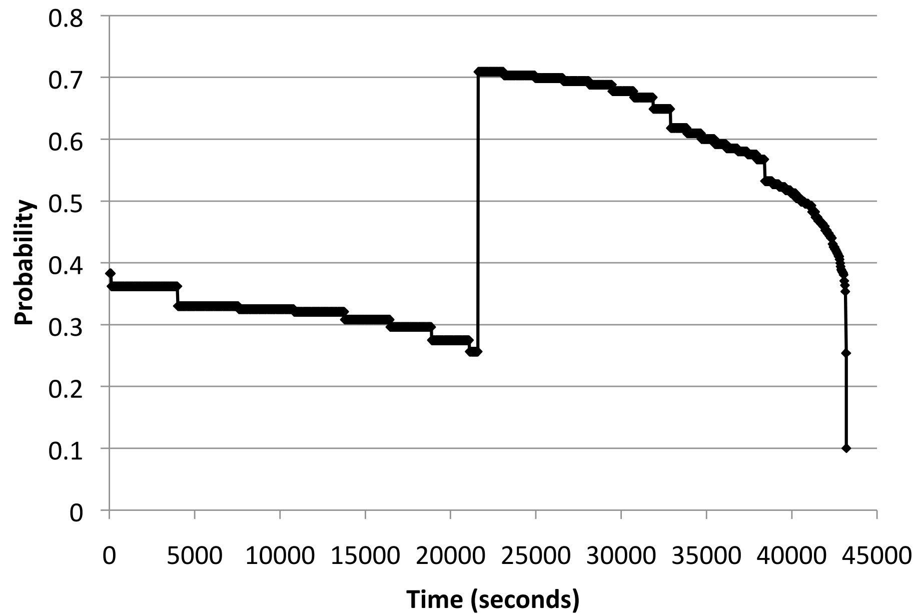

Figure 5.4: VARQ probability trajectory for a *4*-node, *1*-hour job in the "dque" on NCSA TeraGrid machine

allocation cost from 86400 node-seconds to 9480 node-seconds. VARQ, at present, reports only the maximum possible allocation overhead since it does not currently attempt to estimate at what time before the deadline the job is likely to begin executing. We believe a "best guess" in addition to the worst case allocation loss is possible, however, and we are pursuing it as part of current efforts.

Notice also that the probability trajectory associated with a particular VARQ reservation may indicate that there is no submission time corresponding to the

user's specified success probability. Returning to the example, if the user had specified a desired success probability greater than 0.7, the system would have responded by indicating that no reservation is possible. This condition is analogous to the circumstance in which a "hard" reservation is denied because a conflicting reservation has already been made. In addition, to handle the possibility that a change-point occurs during a reservation period, VARQ continually generates new probability trajectories while waiting to submit a job.

In sum, VARQ exploits the slowly changing nature of QBETS bounds estimates to determine when in the future a job should be submitted so that it will be running at a specific deadline. Because QBETS estimates are upper bounds, the job may start earlier than the desired time and simply wait, incurring an extra allocation charge while it does. VARQ allows the user to control this cost explicitly by specifying a target success probability that it will try to honor with the minimum potential cost.

5.3 Experiments and Results

To explore the efficacy of VARQ, we report results from a series of empirical experiments conducted using the machines listed in Table 5.1. Each of these machines is currently in production use by a shared, and potentially competing,

user community. To the best of our knowledge, our user login and account specification received "typical" treatment on each machine (with one possible caveat discussed below), and we did not inform the relevant system administrators of the experiments. We also conducted a simulation experiment to understand the effects of multiple users submitting VARQ jobs as a preliminary investigation of its potential generality. Without cooperation from site administrators, however, we felt it ill-advised to conduct "stress" tests involving multiple and frequent VARQ requests in live settings in which unsuspecting users could be exposed to unforeseen system response.

5.3.1 Efficacy Experiments and Apparatus

Table 5.1 describes the characteristics of the machines we chose to use for our experiments. We chose these machines for a number of reasons: Each machine is supported by QBETS, has a number of active users (although some machines are observed to be busier than others), and provides us the low-level ability to instrument the submission and tracking of job status necessary to perform an actual experiment and gather meaningful results.

In each experiment a submitting process formulates a job, and then selects a specific time in the future when the job needs to be running, and a probability of success. Ideally, we would have liked to perform this experiment for all job

	.50		
machine	predicted	actual	count
datastar	**0.52**	**0.42**	**36**
ucteragrid	0.76	0.98	45
dante	0.90	0.80	61
cnsidell	0.54	0.71	62
ncsateragrid	0.53	0.74	23
iuteragrid	0.80	0.88	24
ornlteragrid	0.88	1.00	39
all	0.71	0.79	290

Table 5.2: Average predicted success probability and actual success fraction for VARQ reservations with minimum success probability of 0.50.

	.75		
machine	predicted	actual	count
datastar	**0.75**	**0.71**	**17**
ucteragrid	0.86	1.00	45
dante	0.93	0.78	59
cnsidell	0.76	0.88	66
ncsateragrid	0.76	0.77	13
iuteragrid	0.81	1.00	22
ornlteragrid	0.87	1.00	37
all	0.83	0.89	259

Table 5.3: Average predicted success probability and actual success fraction for VARQ reservations with minimum success probability of 0.75.

sizes, with a large number of future deadlines, and for a multitude of success probabilities. However, since the experiments run in real time, and the delays on these machines can be substantial, exploring every reasonable combination of these factors is infeasible.

We have set up one machine at our host institution to act as a single point where all experiments are launched. On this machine, we run *submitting processes*

	.95		
machine	predicted	actual	count
datastar	0.97	0.00	1
ucteragrid	0.96	0.96	48
dante	**0.96**	**0.85**	**61**
cnsidell	0.00	0.00	0
ncsateragrid	0.00	0.00	0
iuteragrid	0.97	0.94	18
ornlteragrid	0.97	1.00	58
all	0.96	0.93	186

Table 5.4: Average predicted success probability and actual success fraction for VARQ reservations with minimum success probability of 0.95.

designed to act as users that make reservation requests to VARQ for the HPC targets listed in Table 5.1.

Each process targets a specific machine and queue. When it is initiated, the job is passed a specific success probability and deadline (expressed as a duration until a reservation should be made) as parameters. It begins by composing a job for submission through VARQ using a randomly selected node count and run time from the following sets: either 1, 4, 8, 16, or 32 nodes and either 600, 1800, 3600, 7200, or 14400 seconds of run time. Once the submit process has crafted a job, it queries VARQ regarding the possibility of attaining a virtual reservation using the newly minted job and the deadline and success probability originally specified when the submit process was initiated. If, after computing the necessary probability trajectory for the reservation, VARQ cannot find a submission time in

the future that will satisfy the reservation specification at the desired probability levels, it reports "unable to make reservation" to the submission process which sleeps for 15 seconds, composes a new random job, subtracts 15 seconds from the deadline (so that it targets the same point in time in the future), and retries. The process continues to retry every 15 seconds until VARQ accepts the reservation or until the deadline is decremented to zero. If the latter conditions occurs, the submission process resets the deadline it is attempting to its originally specified value (thereby picking a new target time in the future for a reservation) and continues to retry. Once the submit process successfully makes a reservation with VARQ, it then waits until shortly after the deadline has expired and starts again, attempting a new reservation one deadline duration into the future.

The intention of the protocol is to model a user who wishes to obtain a reservation that starts at a specific point in time, and who is willing to re-query the system in the event VARQ is unable to grant the request. It has the effect, however, of making the time between attempted reservations more or less equal. We do not believe this induced periodicity affects the results negatively, particularly since we observed a fair amount of "drift" in the experiment cycle for each process over the entire experimental period.

For instrumentation purposes, the experimental apparatus determines the success or failure of a VARQ job in meeting its deadline, and the actual allocation

overhead incurred, by searching through the batch scheduler logs on the target machine *post facto*. We considered adding an instrumentation facility to VARQ itself to allow users to query the success history of their own reservations. Such an extension would increase the intrusiveness of VARQ substantially, however, since in its current form the only component that requires access to the local batch scheduler logging information is QBETS.

On the launching machine, we run an experiment process for each of three required probabilities (0.5, 0.75, and 0.95) and the same deadline duration (to speed the time to results). We stagger the start times of these processes so that they do not all target exactly the same moment as a deadline. Also, we for the sake of alacrity, we have chosen a deadline duration of 21600 seconds (six hours), both to improve the number of completed experiments, but also because we felt that six hours the shortest reasonable lead time a user would normally expect to be able to make static advanced reservations. As shown in Figure 5.2, the percentile time series for these machines is typically stable for several days. If a short reservation is possible, longer ones should be more likely within the confines of this stability.

5.3.2 Efficacy Results

To determine the efficacy of the VARQ system, we compare the percentage of successful VARQ attempts to the specified success probability. For example, a submit process attempting to make VARQ requests with 0.5 success probability, should have at least 50% of the submissions accepted by VARQ start before their specified deadlines.

In Tables 5.2, 5.3, and 5.4, we compare the target success probabilities with those we observed across all machines. Each row corresponds to a specific machine (we used the default queue in each case). For each of three different success probabilities (0.5, 0.75, and 0.95) we show three columns of numbers: the average expected success probability used by VARQ, the actual fraction of jobs accepted by VARQ that met their deadlines, and the number of accepted jobs. Recall that VARQ uses the latest time in its probability trajectory that *exceeds* specified success probability as a way of reducing potential allocation overhead. In some cases, this probability may be quite a bit larger than that specified, especially when the machine is lightly loaded. For example, using a specified 0.5 success probability on *ornlteragrid*, VARQ submitted a job when it "saw" predicted success probability, on the average, of 0.88 in the probability trajectories it computed. 100% of the 39 jobs it submitted in this category met their deadlines (as shown in columns 2, 3, and 4 of Table 5.2 in the row for *ornlteragrid*).

These results indicate that VARQ, in the mode we have tested it, is quite successful. Of the 21 test cases (7 machines at 3 target probabilities each) only *dante* at the 0.95 target level and *datastar* at the 0.5 and 0.75 were probabilistic failures (shown in bold face in the table). There were several instances, however, where VARQ refused to accept any reservations, or only accepted one. These are not failures in the sense that the user (the submission process in our case) did not experience a different quality of service than the one VARQ agreed to deliver.

Returning to the observed failures, in *dante*'s case, of 61 jobs accepted by VARQ, with an average predicted success probability of 0.96, only 0.86 (52 jobs) successfully met their deadlines. We provide a more probing analysis of this case also in the next section. For *datastar*, however, the problem was that our job submissions were being assigned (accidentally) to an account used for educational purposes and not research. Apparently jobs submitted to this account receive degraded scheduling priority in comparison to the "average" research user tracked by QBETS. We discovered this anomaly only in *post mortem* analysis of the experiments. At the time of this writing, we have re-initiated the *datastar* experiment under the correct account, and the results for the small number of attempts show success. For the sake of uniformity, however, we felt it unwise to replace the *datastar* numbers with the new data since it was not part of the original experi-

mental run and also because the number of attempts so far is too small to yield a meaningful inference.

For all other tests, however, in which the count of jobs attempted is not 0 or 1, the observed fraction of successes exceeds the average predicted success probability. These results combine to show that VARQ is conservative with respect to success probability. Generating Tables 5.2, 5.3, and 5.4 required 457 hours of wallclock time. We initiated the experiments at 11:30 AM on February 15th, 2007 and terminated them at 12:48 PM on March 6th, 2007. In many cases, despite retrying every 15 seconds, VARQ could not identify a single instance in a probability trajectory that it predicted would result in a successful reservation over the entire experimental period.

5.3.3 Allocation Overhead Results

In Table 5.5 we show the effects of VARQ on the allocation charges incurred during the experiment described above. Organized in a way similar to Tables 5.2, 5.3, and 5.4, each row corresponds to a specific machine, and each of the three major columns represents results for different specified success probabilities (0.5, 0.75, and 0.95 respectively). In Table 5.5, each major column shows the total allocation required to execute the jobs in that category (denoted *required*), the actual allocation used by VARQ in that category (denoted *used*) and the ratio of

machine	.5			.75			.95		
	req.	used	ratio	req.	used	ratio	req.	used	ratio
datastar	14	22	1.58	8	9	1.16	0	0	0.00
ucteragrid	1038	1120	1.08	877	956	1.09	563	1221	2.17
dante	884	944	1.07	604	735	1.22	831	2040	2.45
cnsidell	257	636	2.48	60	212	3.53	0	0	0.00
ncsateragrid	58	127	2.17	28	84	3.00	0	0	0.00
iuteragrid	82	82	1.00	91	91	1.00	110	258	2.34
ornlteragrid	464	473	1.02	628	640	1.02	83	94	1.13
all	2797	3405	1.22	2295	2725	1.19	1587	3612	2.28

Table 5.5: Non-VARQ (*req.*) and VARQ (*used*) allocation costs and their ratio. Cost units are node-hours.

allocation used to allocation required (denoted *ratio*). The units of allocation in this table are node-hours. For example, columns 2, 3, and 4 of the row marked *ornlteragrid* show that the VARQ reservations submitted with a 0.5 success probability required 464 total node-hours of occupancy to execute the work in all jobs and 473 node-hours for that occupancy and the additional cost when jobs started early under VARQ. The ratio of 1.02 indicates the cost factor associated with the use of VARQ. That is, the submission process in this experiment "spent" 1.02 times as much allocation to obtain VARQ reservations as it would have spent had it simply submitted the jobs (without reservations).

From the table, the allocation overhead penalty VARQ introduces varies from machine to machine. On the *cnsidell* machine, for example, VARQ reservations at the 0.5 probability level cost the allocation almost 2.5 times the non-reserved outlay compared to a cost factor of 1.08 on *ucteragrid*. This variability is consistent

with our experience in developing QBETS previously in that each of the machines in this study displays a unique queue delay response profile. Also confirmed is the notion that greater certainty (in terms of higher success probability) implies a greater allocation cost since the cost factor increases monotonically from left to right in each row.

5.3.4 Generality Experiments and Apparatus

To be able to test the effectiveness of VARQ when a sizeable fraction of user-offered jobs are under its control, we constructed a faster-than-real time, trace-based simulator which uses the same VARQ infrastructure we used for the empirical experiment and the Maui [44] batch-scheduler running in simulation mode. The Maui scheduler is the actual scheduler deployed at many of the sites we tested empirically (See Table 5.1). Maui includes a simulation capability that allows input job workloads to exercise a given scheduler policy so that potential performance effects can be identified prior to deployment. Because we did not have access to the specific scheduler policy files at each site, we chose Maui's default policy which is first-come-first-served with backfilling [44, 34] and a processor node count set to 272, which is the total number of nodes in the Datastar machine at SDSC, where each node has 8 processors.

We hasten to clarify that we do not claim the performance results generated by this simulator are representative of any actual system hence we did not use it to investigate the effectiveness of VARQ. However, the question of how well VARQ performs when the fraction of offered workload controlled by VARQ increases is one we believe must be considered. To do so, we use the simulator to compare VARQ performance over repeated experiments where we vary only the fraction of jobs that use VARQ. If VARQ is to be a generally useful methodology, it must be able to support an appreciable fraction of the workload experienced without breaking down or adversely impacting competitive users outside the prioritization specified in the local scheduler policy.

The simulator takes a workload trace (we chose the *datastar* "normal" queue since it seems particularly active), a fraction of jobs that should use VARQ and a success probability. Next, the simulator chooses regular time periods (six hours apart) within the trace indicating times of advance reservation start deadlines. This mode of operation represents the worst case where the given percentage of jobs request reservations starting at same time every six hours. Jobs are considered in submission order, and a job is selected to be considered a VARQ job randomly, but in proportion to the specified fraction (e.g. if the fraction is 0.10 each job has a 10% chance of being converted into a VARQ job). If the job is selected, it is

Percentage VARQ jobs	.5			.75			.95		
	pred.	act.	count	pred.	act.	count	pred.	act.	count
10	0.54	0.66	359	0.77	0.72	60	0.97	0.91	11
50	0.55	0.60	1869	0.77	0.68	1108	0.96	0.84	194
95	0.68	0.64	1960	0.43	0.81	939	0.98	0.97	48

Table 5.6: Average predicted success probability (*pred.*) and actual observed success fraction (*act.*) for VARQ jobs in simulation.

presented to VARQ for execution at the next six hour deadline with the specified success probability.

5.3.5 Generality Results

The results of our simulation experiment are shown in Table 5.6. Note that although there are times when we failed to achieve the minimum expected success percentage using VARQ, in most cases VARQ was able to acquire a success percentage very close to the expected success percentage of reservations. Predictably, as the fraction of VARQ jobs increases and the success probability increases, VARQ's success rate decreases. However, in many cases the results are surprisingly close, given the extreme nature of the simulation. For example, if 10% of the jobs are VARQ jobs and they target the same deadline with a desired success probability of 0.95, the observed success fraction is 0.91. Only when 95% of the jobs are VARQ jobs and the desired success probability is equal to 0.75 do the simulations show VARQ's quality of service guarantees breaking down. Note

that when 95% of the jobs are VARQ jobs and the desired success probability is either 0.5 or 0.95 VARQ almost succeeds. In the former case, the conservativeness of QBETS predictions furnishes VARQ with enough "slack" in the estimate of the 50th percentile so that the predicted probability (0.68) is only slightly larger than the observed success fraction (0.64). In the latter case, QBETS is able to find few instances where it predicts the probability to be 0.95 or greater. A closer analysis of the simulation trace reveals that the 48 VARQ attempts in this case only occurred during period of light workload in the job trace we used. Thus we observe in this example there is a regime between the extremes of 0.5 and 0.95 success probability where VARQ clearly fails to operate. We believe this effect is general, but the precise failure regime will be site specific.

At a high level, these results indicate that VARQ is both likely to offer a larger user community valuable functionality without degrading resource performance or utilization. In the simulations, each VARQ job had its run time *increased* to cover the possibility of an early start. Either the simulated machine was under utilized by the original (non-VARQ) workload, in which case VARQ increases the utilization perceived by the system administration, or the machine was originally over committed, in which case VARQ does not cause utilization to be lost. Moreover *all* of the simulations executed in approximately the same simulated time interval as did the VARQ-free simulation (not shown). Thus the amount of work accom-

plished with active VARQ jobs is approximately the same as when no VARQ jobs are present. We present these results in order to provide evidence that a more aggressive field test of VARQ is warranted as its general use (even in the worst case) appears relatively benign.

5.3.6 Discussion

The results presented in the previous subsection show that VARQ implements a new advanced reservation abstraction for HPC users. Moreover, the abstraction is virtual. Existing batch systems, governed by complex and hidden local scheduling policies, do not need to change in any way and, in particular, do not need to agree to support any form of user-initiated reservation mechanism for VARQ to function. Finally, the virtualization is statistical. VARQ uses predictions generated by QBETS to “manufacture” a reservation without local infrastructure support. As a fortuitous side-effect, each VARQ reservation can be characterized by success probability that is conservative. Users know the minimum success probability associated with each of their reservations. Together, these features enable VARQ to achieve a functionality first hypothesized as being useful almost a decade previously [10] and for which we believe there will be substantial demand.

VARQ also offers several capabilities that would otherwise be difficult to implement as standardized batch queue software mechanisms:

- It trivially implements a zero-overhead "best effort" reservation. In this mode, a user specifies a deadline and a success probability, but is willing to tolerate having the submitted job start before the deadline.
- It allows users to control what they pay in allocation as a function of how precisely they wish to have their deadlines met.
- It allows VARQ reservations to be combined from independent sites with predictable joint probabilities of success.

The first feature is simply a function of when a user job actually begins doing work after it has successfully be allocated a set of processors. If a user does not want to pay the allocation overhead necessary to wait until her deadline, her job need only begin executing immediately instead of 'spinning" until the deadline. In the current prototype, this spinning or waiting is implemented in the application itself. It is trivial to wrap the application in a script to avoid the need for user modification of the program. Either way, however, the user can control the cost at the time the job begins node occupancy. This leads to the second new capability, which is specifically that a user of VARQ has explicit knowledge of the "costs" involved in acquiring an advance reservation. Knowing both the probability with which a reservation will be granted and the maximum over-allocation cost allows users or higher level work planning systems to potentially apply customizable

cost models to resource selection and scheduling processes. Currently, finding the "cost" of hard advance reservation allocation on existing systems is ill specified, non-standardized, and often simply impossible. We hope that VARQ's ability to both expose and control the success probability and over-allocation costs of virtual advance reservations will lead to new research in workflow planning and distributed resource scheduling.

Finally, because each VARQ reservation is associated with a success probability, it is trivially possible to combine reservations to meet a specific reliability target as long as the user can take advantage of which ever resource ultimately is delivered first. For example, if it is possible to obtain two different VARQ reservations for the same point in time, each with a success probability of 0.9, and the machines behave independently with respect to queue delay (which they almost certainly do at present), the probability of getting one or the other or both is 1.0 minus the probability that they will both fail. That is, the joint probability of success in this example is at least 0.99. While users may not take advantage of this simple approach manually since it involves canceling one of the submissions to avoid even greater allocation cost, in grid settings, where meta-schedulers can manage this complexity automatically, the possibility is intriguing.

However, the using VARQ for co-allocation induces the reverse effect. That is, the joint probability associated with co-allocated reservations is less than the

probability of either one. However, our future efforts are beginning to show that it is possible to use the “either-or” approach to mitigate this probability decay in a grid setting of sufficient scale. Because VARQ reservations are characterized by explicit success probabilities, using them coherently in combination under programmatic control has great promise that we leave to future work.

5.4 Conclusion

One major hurdle the HPC community has yet solve generally is that of providing users and grid/metacomputing systems the ability to obtain advance reservations. In this Chapter, we describe VARQ, a statistical approach which provides scientists with a familiar mechanism to obtain virtual advance reservations on existing systems, without affecting local site software or administration and scheduling policies. We show through empirical experiment that VARQ successfully obtains resources corresponding to a variety of user requests on real HPC systems in operation, and provide evidence that the introduction of VARQ as a more general tool is likely to be effective and will not cause a substantial impact on resource performance. In the next chapter, we will extend the idea of VARQ to handle the synchronized allocation of resources on multiple, independently managed HPC sites.

Chapter 6

Statistical Co-allocation

6.1 Introduction

In the previous Chapter, we presented a methodology for provisioning HPC resources during a specified time interval atop existing best effort batch systems. Our methodology, VARQ, provides a user with a probabilistic guarantee that their specified resources will be available during their specified time interval, but we focused our verification experiments on independent resources. While this facility is solves a significant problem for users that require an 'advance reservation' service for individual machines, there is a need to a similar functionality across different machines. Further, applications within this class can grow to a size where they require more resources than are available at any one site, or have some application component that always must run at a specific site, but others that can execute elsewhere to improve performance (for example, a data gathering component that

reads data from a weather sensor station needs to execute near that station, geographically, even though the computationally intensive application component can execute elsewhere). These applications are currently impossible to reliably execute on production HPC systems, since most sites do not support general mechanisms that allow the user to, automatically and consistently, guarantee that their application components, distributed across multiple sites, will be granted their requested resource set at or before a fixed time in the future. Generally, this situation is referred to in the literature as the **co-allocation problem**.

In the previous Chapter 5, we introduced a novel system for providing regular HPC users the ability to make *virtual* advance reservations. Our system, termed Virtual Advance Reservations or Queues (VARQ), uses a probabilistic technique to decide when, between the request for a reservation and that reservation's desired start time, to submit a batch job such that the resources are available during the requested time period. Since VARQ is implemented as an overlay on top of existing batch queue software and policies, and uses queue wait time predictions (QBETS [48, 4]), the advance reservations granted have a well defined probability of success. While we showed that VARQ can be used to acquire a single reservation on a single machine with a reasonable probability of success, we note that if we try to make multiple VARQ requests simultaneously, and further require that that they all succeed for the reservation to be useful (thus implementing a

co-allocation service), the success probabilities quickly plummet since the joint probability of all reservations succeeding is the product of the individual reservation probabilities. In this Chapter, we observe that if we are able to relax an input parameter of VARQ, namely the actual site on which a single reservation is to be serviced, we can boost the probability of acquiring multiple VARQ reservations by making multiple VARQ requests simultaneously, discarding those that turn out to be redundant. Our results show that the combination of VARQ with our novel probability boosting techniques (which we term CO-VARQ) allows regular users of production HPC machines to request co-allocated reservations on top of existing, best effort batch queues with a configurable probability of success.

The rest of this Chapter is organized as follows: in Section 6.2, we discuss the methodologies we use to implement CO-VARQ. Finally, we present a description of an experiment we've designed to test the effectiveness of CO-VARQ and costs associated with using the system in Section 6.3. We conclude in Section 6.4.

6.2 Methodology

In this section, we describe the methodology we employ to attempt to solve the co-allocation problem probabilistically, as an overlay on top of existing software and platforms. At a high level, our techniques differ sigificantly from past

approaches in that our methods rely on observation of empirical job, queue and policy behavior instead of attempting to introduce some level of control into the existing systems. Aside from being a much more pragmatic approach, our tactic has shown, thus far, an incredible level of flexibility as the same approach has survived fluctuations in workload, software changes, and even significant scheduler policy adjustment. Next, we briefly describe our system for making virtual advance reservations and describe how we extend the concept to implement a co-allocation service.

In the previous Chapter we describe VARQ, which allows regular HPC users to make advance reservation requests on individual existing batch queue controlled HPC systems, without requiring any change to the local system software or scheduling policies. The benefits of VARQ are clear: adding the ability of regular users to make advance reservations on existing HPC systems without modification to system software, special priviledges, or even notification of the local resource administrators thus allowing for automatic and frequent use. However, there are costs associated with using VARQ.

First, in order to guarantee that a reservation will be serviced during a specified time period, recall that VARQ typically submits a placeholder job that is longer than the actual requested reservation. Although VARQ chooses the latest time to submit in order to minimize this wasted allocation, there were cases where

a VARQ placeholder used up three and a half times the allocation that the job actually required to execute. Again, this (worst case) cost is known at the time of VARQ submission, allowing the user to decide (along with the success probability, discussed next) whether the over-allocation expenditure is less valuable than the ability to get deterministic resource availability. The second "cost" is less obvous. As an input to VARQ, a user must specify a minimum success probability which must be necessarily less than 1. We evaluated VARQ using 0.50, 0.75 and 0.95 as input minimum success probabilities, finding that, for the most part, VARQ was able to supply reservations with *at least* the input level of success. However, or the higher probabilities (0.95 and above), we found many cases where VARQ was simply unable to find a point in the future where it could guarantee with such a high level of probability that a request could be succesfully granted. While this effect is not considered a failure (since upon making such a request, VARQ immediately returns a "not possible" message), the reality that there were many cases where we could not make a reservation with a high level of success was a problem we wished to address. Although we consider the necessity to input a minimum success probability a "cost" of using VARQ, it should be noted that using more traditional hard-coded advance reservation systems does in fact have some success probability associated with it, but this probability is neither known nor is it presented to the user, and so VARQ's ability to return a "not possible at

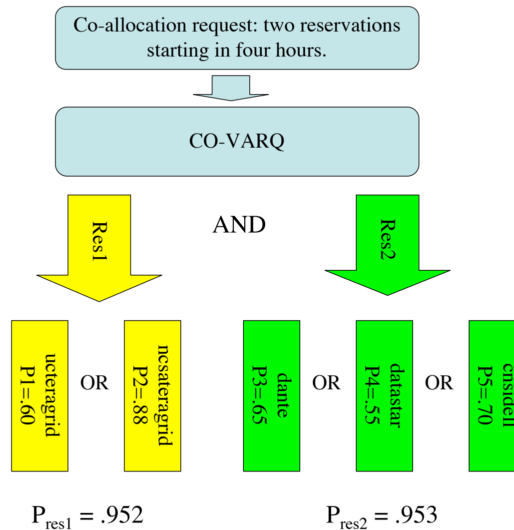

Figure 6.1: Steps involved when a CO-VARQ request (consisting of two simultaneous advance reservations) is supplied. Here, CO-VARQ executes two VARQ processes in order to procure one reservation, and three VARQ processes to procure the other, resulting in an overall probability of success equal to $0.952*0.953 = 0.91$.

the input level of success" message, upon reservation request time, can be seen as added functionality.

6.2.1 CO-VARQ Overview

When considering the use of VARQ to implement a co-allocation service, we quickly discovered that requiring more than one VARQ request being succesfully serviced had a drastic impact on the success probability that could be achieved. Consider the following example; a user requests a single VARQ reservation on

machine A with a minimum success probability of 0.95. Next, the user wants to add a second reservation on machine B at the same level of success. Independently, each reservation has a probability of 0.95 of being serviced successful, but the probability that *both* are successful is the product of the two, namely 0.903. Adding a third reservation results in a joint probability of 0.857, etc. One approach is to fix the joint probability at the user specified level and reverse solve for the independent VARQ input probabilities. For example, we allow the user to specify that they wanted two simultaneous VARQ reservations with a joint probability of success equal to 0.95. Thus, each reservation needs an independent success probability of $0.95^{1/2} = 0.975$. Three reservations require independent probabilities of $0.95^{1/3} = 0.983$, etc. Although this method is sound, we find that VARQ is generally unable to satisfy requests at theses minimum success levels unless the deadline is generally far in the future (say, a day or more). Since we wish to provide users the ability to make reservations with as little as a few hours of lead time, we use a different approach to "boost" the success probabiliies of independent VARQ requests.

Because VARQ reservations are characterized with success probabilities, we can combine them in order to "boost" the probability of any one of the requests being successful. For example, in our above example, if the user did not care *where* a resevation was serviced, just that it was serviced at the right time with

a specified success probability, then we can submit more than one VARQ job on multiple machines with each request having a lower level of success than the overall input level. For example, consider a user with a roaming allocation who requires a single reservation on either machine A or machine B. If VRQ could service either of these reservations with a minimum probability of 0.90, then the probability P_{either} that one or the other is succesfully granted is:

$$P_{either} = 1.0 - 0.1^2 = 0.99 \tag{6.1}$$

That is, if we assume that the success of each VARQ reservation is independent, the failure probability for each VARQ reservation is 0.1, and so the joint probability they will both fail is 0.01. Although we can use this technique to "boost" the probability of an individual reservation request, we focus here on using it to implement a co-allocation service, since the ability to provide simultaneous requests is a super-set of a the ability to provide one.

CO-VARQ uses the both the independent request probability reverse solving technique and the probability "boost" technique described above to provide a co-allocation service. First, we define a set of resources from which CO-VARQ can attempt to make a VARQ request. Next, CO-VARQ reads the co-

allocation request specification which has the following form, where $N = nodes$, $W = maxWallTime$, $D = startDeadline$:

```
N1 W1 D1
N2 W2 D2
...
```

CO-VARQ remembers the number of allocation requests in total ($COUNT$). In addition, the user specifies a single minimum success probability (P_{all}) that is understood to defined as the joint probability that *all* reservations listed in the above specification are successful. Next, the CO-VARQ engine reverse solves the minumum success probability that each independent VARQ request must satisfy:

$$P_{indep} = (P_{all})^{1/COUNT}$$

CO-VARQ then iterates through all of the resources in it's resource set, generating a table of probabilities for each specified reservation on each resource. Finally, we iterate over each input reservation, drawing the VARQ request from the resource pool with the highest expected probability of success, completing when we have either exahusted our resource set, or when each input reservation has a joint probability high enough to satisfy P_{all}. The resulting VARQ "plan" is then acted upon, with CO-VARQ making as many VARQ requests as it determined were necessary to acquire the specified reservations.

In Figure 6.1, we show a typical example of a single CO-VARQ request. In the example, a user requires two ($COUNT = 2$) reservations be made, each starting at the same time, four hours from the time of the request. As described earlier, the user is allowed to additionally specify their desired minimum success probability, in this case we've chosen to depict the default of $P_{all} = 0.90$. CO-VARQ first determines the minimum probabilities that each reservation must satisfy ($P_{indep} = (P_{all})^{1/COUNT} = 0.95$). Finally, CO-VARQ begins querying VARQ to determine how many VARQ requests need to be submitted per reservation in order to meet P_{indep}. In the example, CO-VARQ found that for **Res1**, two VARQ requests must be submitted (one on **ucteragrid** and one on **ncsateragrid**) resulting in a joint probability that *one* of the VARQ requests is successful equal to $P_{res1} = 0.952$. For **Res2**, CO-VARQ found that three VARQ requests needed to be made to end up with a joint probability of $P_{res2} = 0.953$. Once this "schedule" of requests has been decided, CO-VARQ submits five total VARQ requests and informs the user as to which reservations are successfully acquired during the requested time period, cancelling those that are redundant once one VARQ request from each reservation has been successfully obtained.

Machine	Processors	Batch Software	Description
datastar	2176	Load Leveler	SDSC IBM PowerPC Production Compute Cluster
ucteragrid	316	Torque/Maui	UC/ANL IBM/Intel Compute Viz Linux TeraGrid Cluster
cnsidell	256	Torque/Maui	UCSB NanoScience Research Linux Cluster
ncsateragrid	1744	Torque/Maui	NCSA IBM/Intel Compute Linux TeraGrid Cluster
sdscteragrid	524	Torque/Maui	SDSC IBM/Intel Compute Linux TeraGrid Cluster

Table 6.1: HPC machines used in CO-VARQ empirical experiments.

6.3 Experiments and Results

To test the effectiveness of CO-VARQ in practice, we have designed a simple experiment that exactly models the behavior potential users. We chose five production and research HPC machines to use for this study that we believe fairly represent a set of machines that a typical user might have access to simultaneously. In order for a machine to be included in the set of supported systems, the only requirement is that the QBETS batch queue prediction system is currently monitoring the system. Otherwise, VARQ and CO-VARQ operate entirely on top of the existing software stack, and requires no communication with the administrators or special access. In Table 6.1, we describe the machines we used for this study. The UC/ANL TeraGrid, NCSA TeraGrid, SDSC TeraGrid and SDSC Datastar systems are all production qualit super-computers operated by

NSF/DOE centers, while the CNSI Dell system is a large University cluster operated at our home institution.

The experiment is designed as follows: we first pre-determine a set of co-allocation request parameters **COUNT, N, W, and D**, along with a required minimum success probability that all reservations will succeed (P_{all}). Next, we execute a script that calls the CO-VARQ system once every $D + W$ hours, so that we only ever have one CO-VARQ experiment running at a time, avoiding the possibility that our CO-VARQ experiements will interfere with one another. CO-VARQ will attempt to compute the "plan" of VARQ requests necessary to fulfill P_{all} and if successful, will implement the plan by calling VARQ and monitoring the progress of each reservation request. If CO-VARQ cannot compute a schedule that fulfills P_{all}, it gives up and waits ten minutes before trying again.

The experiment is designed to accomplish several goals. First, we wish to verify the correctness of CO-VARQ by comparing the percentage of successful CO-VARQ requests to the specified minimum success probability. Second, the experiment allows us to inspect, for successful CO-VARQ requests, how much excess allocation is used in order to achieve co-allocated resources. Finally, the experiment is designed to execute in a real production environment using a variety of HPC resources and user workload behaviors over a relatively long period of time.

Job Params (N,W,D)	P_{all}	$P_{expected}$	$P_{achieved}$
4, 1hr, 4hr	0.90	0.942	0.901
4, 1hr, 12hr	0.75	0.877	0.920
16, 1hr, 12hr	0.75	0.866	0.957
48, 1hr, 12hr	0.75	0.836	0.906

Table 6.2: Minimum success probability (P_{all}), calculated expected probability ($P_{expected}$), and actual achieved probability ($P_{achieved}$) of four CO-VARQ trials.

In total, the experiment ended up using resources from six HPC sites over a six month time period (Sep. 2007 - Mar. 2008).

We performed the experiment with various values of N, W, D and P_{all} (each N,W,D,P_{all} tuple is considered a trial), and report the percentage of total CO-VARQ requests that were successful along with the total over-allocation costs assoctiated with each trial.

The results of our experiment are shown in Table 6.2. In the table, each row depicts the percentage of co-allocation requests that resulted in a successful provisioning of the specified resources. In the first column, we describe the 'shape' of the jobs that were specified in terms of a tuple (nodes, job execution time, and start time of the co-allocated jobs). In the second column, we show the specified minimum success probability that was given to CO-VARQ when the co-allocation request was made. The third column shows the average actual probability of success that CO-VARQ computed given the conditions that existed when the request

was made. The fourth column shows the percentage of co-allocation requests that CO-VARQ was able to successfully provision.

In sum, the results indicate that for all of the trials we have completed thus far, CO-VARQ is able to successfully co-allocate resources for jobs at at least the target minimum success probability, and in each case actually exceeds the expected probability of success ($P_{achieved}$ is greater than both $P_{expected}$ and P_{all}). This result indicates that while CO-VARQ is able to successfully co-allocate resources at the specified target minimum success probability, our calculations must be somewhat overly conservative since we're achieving a higher percentage of success ($P_{achieved}$) than we should expect ($P_{expected}$). We believe that the conservative results are a residual effect of using the conservative bound estimates on job queue delay that QBETS provides. These experiments are on-going, with a wider range of target minimum success probabilities (0.50 and 0.90) and job walltimes (4 and 8 hours).

6.4 Conclusion

While existing batch queue systems efficiently manage local HPC workloads, the software and, more importantly, the local scheduling policies, are not designed to support the co-allocation of multiple reservations across site boundaries. Although there has been significant work showing that meta-schedulers can solve the

co-allocation problem, they all require that the local site give up some amount of control over how their resources are manage, which has thus far made these global scheduling systems infeasible in practice. In this Chapter, we present CO-VARQ, a system that uses a statistical technique to provide co-allocated reservations using the existing best effort batch queues, without requiring any modification to local site software, policy or control. We show that, in practice, CO-VARQ was able to successfully reserve the target percentage of co-allocation requests on five production and research HPC machines currently in operation today.

Chapter 7

Related Work

7.1 Backgound and Related Work

There is a great deal of literature in the fields of grid and distributed systems surrounding the problems apparent in characterizing, managing and predicting the performance response of resources in a heterogeneous, distributed setting. Here, we cover two major areas related to performance dynamism of resource availability (uptimes, downtimes, and failures) and provisioning (predicting, modeling and characterizing provisioning delay).

7.1.1 Resource Availability Dynamism

When high-performance resource pools are composed of very large numbers of heterogeneous resources, the amount of time that individual resources remain available once acquired can be highly variable. Designing the next-generation of

Grid applications that can run in such an environment requires an accurate model of resource failure behavior. A great deal of previous work [46, 37, 25, 33, 38] has studied the problem of modeling resource failure (or equivalently resource availability) using statistical techniques. As Plank and Elwasif point out in their landmark paper [54], however, most of these approaches assume that the underlying statistical behavior can be described by some form of exponential distribution or hyperexponential distribution [38]. In addition, they go on to note that despite their popularity, many of these modeling techniques do not accurately reflect empirical observation of machine availability. Other work has been done showing that a Weibull distribution is an appropriate model for various resource availability data [68, 31] but neglects to provide a detailed analysis of fitting and model verification stages.

Exponential distributions have been studied extensively in fault tolerant computing settings [64, 41, 54, 55, 42]. More recently, peer-to-peer systems have used exponential distributions to as the basis of their availability assumptions [62, 69, 70]. However, models of resource lifetimes based on the exponential distribution have been shown to be inaccurate, though model inaccuracies may not have a significant negative impact on the application of the model, depending on specific model usage [42]. In other contexts such as process lifetime estimation [30]

and network performance [52, 51, 66, 9, 39] researchers often advocate the use of "heavy-tailed" distributions, especially the Pareto.

We performed our own studies of Grid-resource availability [47, 3, 49] and discovered that the use of more accurate parametric models improve methodologies that rely on models to make optimization decisions at runtime. In addition, in [3], we show that while parametric models can be used to make somewhat accurate machine uptime predictions, non-parametric techniques have some key characteristics that make them an attractive choice when predicting future events. First, the techniques we studied required fewer data points to make accurate predictions. Parametric modeling techniques tend to characterize the entire population in order to make a prediction for an individual event, while the non-parametric techniques are better at characterizing only the aspect of the population that is required to make a prediction. Second, we note that, while the parametric techniques require us to choose a family of statistical distributions in advance with which to model our data, the non-parametric techniques impose no such requirement. This feature is important due to observed anecdotal evidence that indicates that resource uptimes vary widely between sets of resources, and even over time within the same resource. We found that, while we could discover an accurate parametric model of resource uptimes for a static data-set, the model does not necessarily hold up over

time or between disparate sets of resources, while the non-parametric techniques have proven robust in the face of ever changing underlying behaviors.

7.1.2 Resource Provisioning Dynamism

The second set of previous related literature addresses the problems imposed by substantial and variable batch queue job delay observed on HPC resource. Previous work in this field can be categorized into two groups. The first group of work belongs under the general heading of the scheduling of jobs on parallel supercomputers. In works by Feitelson and Rudolph [19, 20], the authors outline various scheduling techniques employed by different supercomputer architectures and point out strengths and deficiencies of each. The prevalence of distributed memory clusters as supercomputer architectures has led to most large scale sites using a form of "variable partitioning" as described in [19]. In this scheme, machines are space-shared and jobs are scheduled based on how many processors the user requests and how much time they specify as part of the job submission. As the authors point out, this scheme is effective for cluster-type architectures but leads to fragmentation as well as potentially long wait times for jobs in the queue.

The second field of previous work relevant to our work involves using various models of large-scale parallel-job scenarios to predict the amount of time jobs spend waiting in scheduler queues. These works attempt to show that batch-

queue job wait times can be inferred under the conditions that one knows the length of time jobs actually execute and that the algorithm employed by the scheduler is known. Under the assumption that both of these conditions are met, Smith, Taylor and Foster introduce in [60] a prediction scheme for wait times. In this work, the authors use a template-based approach to categorize and then predict job execution times. From these execution-time predictions, they then derive mean queue delay predictions by simulating the future behavior of the batch scheduler in faster-than-real time. In practice, however even when their model fits the execution-time data well, the mean error ranges from 33% to 73%.

Downey [11, 12] uses a similar set of assumptions for estimating queue wait times. In this work, he explores using a log-uniform distribution to model the remaining lifetimes of jobs executing in all machine partitions as a way of predicting when a "cluster" of a given size will become available and thus when the job waiting at the head of the queue will start. As a base case, Downey performs a simulation which has access to the exact execution times of jobs in the queue, plus knowledge of the scheduling algorithm, to provide deterministic wait time predictions for the job at the head of the queue. As a metric of success, Downey uses the correlation between the wait times of the head jobs during the base case simulation and the wait times experienced by head jobs if his execution time model is used.

Both of these approaches make the underlying assumption that the scheduler is employing a fairly straightforward scheduling algorithm (one which does not allow for special users or job queues with higher or lower priorities), and also that the resource pool is static for the duration of their experiments (no downtimes, administrator interference, or resource pool dynamism).

Our work differs from these approaches in two significant ways. First, instead of inferring from a job execution model the amount of time jobs will wait, we make job wait time inference from the actual job wait time data itself. The motivation for why this is desirable stems from research efforts [8, 29], which suggest that modeling job execution time may be difficult for large-scale production computing centers. Further, making inference straight from the job wait time data, we avoid having to make underlying assumptions about scheduler algorithms or machine stability. We feel that in a real world scenario, where site scheduling algorithms are rarely published and are not typically simple enough to model with a straightforward procedure, it is unlikely that valid queue wait-time predictions can be made with these assumptions.

Second, our approach differs in the statistic we use as a prediction. Most often, researchers look for an estimator of the expected (mean) wait time for a particular job. Our approach instead uses bounds on the time an individual job will wait rather than a specific, single-valued prediction of its waiting time.

We contend that the highly variable nature of observed queue delay is better represented to potential system users as quantified confidence bounds than as a specific prediction, since users can "know" the probability that their job will fall outside the range. For example, the information that the expected wait time for a particular job is 3 hours tells the user less about what delay his or her job will experience than the information that there is a 75% chance that the job will execute within 15 minutes.

Our contribution of a **V**irtual **A**dvanced **R**eservation system for **Q**ueues (VARQ) is designed to function in an administrative environment that is typical of science and engineering computing centers serving users with potentially competing resource demands. In this section, we describe the general characteristics of these HPC settings and discuss other research projects that are germane to our effort. VARQ's function depends critically on QBETS [48]. While developing and deploying QBETS in a number of University, National Science Foundation (NSF), and open Department of Energy (DOE) centers, we observed several features common to the way these systems are managed that, in part, make the success of VARQ possible.

7.1.3 Co-Allocation of Distributed Resources

Over the last decade, there has been a great deal of interest in the concept of grid computing [1, 22] (originally termed "metacomputing" [58]), which is essentially the idea of using multiple distributed, heterogeneous resources with minimal global centralized control structures to perform coordinated tasks, such as executing scientific applications. One of the fundamental research hurdles which needs to be overcome to realize a functional grid computing environment is that of resource co-allocation, where multiple disparate sets of resources must be made available simultaneously to some global scheduler. There have been many research efforts indicating that grid "meta-scheduler" systems [6, 13, 14, 24] can increases global system utilization while providing large resource pools, but for the most part these efforts require the use of a centralized, global batch scheduling entity to which *all* jobs (both global and local) are submitted. While this body of work shows a great deal of promise and utility using simulation and closed research environments, the modification and coordination burden placed on local site resource operators has proven to be so severe as to not be applicable in practice. The primary prohibitive modification these systems impose on local site schedulers is that of allowing regular users to make advance reservations in order to support resource co-allocation. Several studies have shown that allowing regular users the right to make advance reservations can have a negative impact on both system utilization and overall

turnaround time for regular jobs. In [59], it was shown that the introduction of a general use advanced reservation system into a normal HPC workload can have a substantial impact on the experience of regular batch users. In this work, the authors convert 10-20% of jobs in historical job traces into "reservation" requests for a upcoming time in the future. Even though the assumption is that the jobs requiring advanced reservations can tolerate some slippage (reservation start time can be delayed if scheduler cannot guarantee resources at the requested time), the average wait time increase for regular batch jobs increased by 9-37% depending on how many reservation jobs were made (10-20% of overall jobs, respectively). In [32], the authors evaluate the impact of advance reservations on a regular batch controlled workload in terms of percentage of reservation requests rejected, slowdown factor of regular jobs (termed "variable" jobs in the paper) and system utilization. Running various simulation experiments, using two reservation algorithms and a real job trace, the authors are able to determine that the introduction of advance reservations increases the queuing delay experienced by regular jobs, but it is difficult to gauge the magnitude of the impact based on the metric used. Finally, researchers in [61] perform a simulation based experiment, using the popular Maui scheduler [44], that attempts to determine the effect of advance reservations and co-allocation requests on regular HPC workloads; the authors suggest methods for minimizing this effect when compared to an alternate tech-

nique for co-allocation in which sites explicitly reserve a specific time every day for explicit meta-scheduler use. Although the authors make a compelling case for the use of advance reservations based scheduling to support cross-site co-allocation of resources, their experiments and conclusions indicate that the introduction of advance reservations have a negative impact on both system utilization as well as queue delay experienced by non-reservation jobs.

As a result of these studies, HPC site operators have been reluctant to adopt the use of general advance reservation or co-allocation systems to support off-site metaschedulers, and are unwilling or unable to relinquish local control of resource scheduling to a global scheduling system. In a particularly relevant work proposing a resource management system for metacomputing [10], the authors acknowledge the fact that local control must be maintained in order for HPC centers to subscribe to metacomputing methodologies, but argue that advance reservations must be supported to fully support co-allocation. Even so, the authors briefly propose a method for making a "best effort" batch submission to support co-allocation without advance reservation, but do not detail the specific mechanisms used to implement their solution, and make it clear that their solution is a temporary situation which will be replaced when sites adopt general advance reservation functionality. However, in the decade since this paper was published, general adoption of advance reservation and/or co-allocation capabilities has re-

mained elusive and shows little sign of becoming enabled on production systems. With growing network capacity, data set size, existence of useful specialized instruments, and workflow application management systems, grid users are once again calling to a solution to the co-allocation problem with renewed voracity.

Chapter 8

Impact and Conclusion

8.1 Impact

The statistical methodologies and techniques that comprise QBETS, VARQ, and CO-VARQ have been implemented as a set of tools and services that have positively impacted the HPC community. The implementation of these techniques not only benefits the community, but also provides valuable evidence for the continuing correctness and accuracy of the techniques as the workloads of HPC systems change through time. Over the past several years, these services have improved the usability of HPC resources for many users, and have allowed some applications that were previously unable to execute in many existing HPC resources to utilize a much larger pool of resources. In addition, we use the data gathered from usage of the services for continual verification of the methodologies.

Currently, QBETS is providing predictions to a growing base of HPC researchers and users around the world. Our batch queue monitoring sensors are gathering real-time batch queue delay data from 16 super-computers, 24 hours a day. From this database of job delay information, the QBETS prediction software is able to constantly generate up-to-date quantile predictions through a number of interfaces. Over the past several years, our records indicate that the QBETS system has been accessed over 3000 times per day, from approximately 2000 unique, non-searchbot Internet hosts. This level of activity indicates that users interested in integrating real-time QBETS predictions into their projects are using a number of interfaces, including a C API (in the form of a UNIX library), UNIX command line tools (for curious users and administrative scripting), a dedicated QBETS Web Service (for integration into existing Web Service based projects), and our own custom QBETS web site [50]. Using these interfaces, scientists have used QBETS in a variety of settings, including the construction and use of HPC site selection hints for users (TeraGrid User Portal [63]), in-advance workflow scheduling for disaster recovery applications (LEAD Project [53]), redundant batch queue resource provisioning for fault-tolerant systems (LEAD/VGrADS [65]), augmentation of TeraGrid meta-scheduling services, and a variety of individual research efforts. As the popularity of QBETS continues to grow, we continue to add more systems to the infrastructure and note that QBETS has been added to the list

of core services that is enabled whenever a new HPC resource is added to the TeraGrid national grid infrastructure.

8.2 Conclusion

Computational scientists require large collections of heterogeneous resources in order to achieve the level of performance necessary to further their science. The concept of "meta-computing" or "grid-computing" outlines a vision where access to computational and storage facilities is as ubiquitous and readily available as electrical power. However, large scale systems are typically not centrally managed and use space-sharing techniques to provide access to site resources, leading to significant and highly variable delay between the time when resources are requested and those resources are made available to the scientist. While some applications can tolerate this resource provisioning delay dynamism, a large set of applications that could potentially use many distributed HPC machines together for a single application execution cycle have been unable to do so.

In this dissertation, we have analyzed over 10 years of resource provisioning delay data in the for of batch queue job delay, and have developed a methodology for predicting, with quantifiable confidence intervals, bounds on the amount of time jobs will wait in queue before beginning execution. We present a detailed

analysis of the performance of the methodology using trace data from 11 large scale HPC machines and find that our non-parametric methodology (QBETS) performs better, and more accurately, than a collection of previously suggested parametric and non-parametric methodologies. QBETS has been successfully used to optimize workflow schedulers and to provide prediction services to the HPC community through a number of interfaces. Building upon QBETS , we present a new abstraction that allows a scientists to obtain a "virtual" advance reservation for queues (VARQ), with a quantifiable probability of success. We implement the methodology as a service for simple integration into existing workflow planners and meta-schedulers that require access to advance reservation capability. We show that VARQ can successfully provision resources, with a specified probability of success, on existing HPC resources without the need to modify local site software of policies. Finally, we present a second abstraction, built upon VARQ, that we use to implement a statistical co-allocation service (CO-VARQ). Using CO-VARQ, we show that co-allocation of resource sets across distributed HPC resources can be obtained using only QBETS and VARQ as it's underlying infrastructure.

Together, these contributions present a new solution to problems that arise from highly variable and significant delays in provisioning HPC resources. They have been implemented as tools that the HPC community has used in practice in a number of settings, allowing scientists simultaneous access to multi-site resources

that have been previously only available individually. Thus, this work provides a framework for both existing and future scientists to leverage resources from extremely large scale, distributed, and highly utilized HPC sites in order to further their science.

Bibliography

[1] F. Berman, G. Fox, and T. Hey. *Grid Computing: Making the Global Infrastructure a Reality.* Wiley and Sons, 2003.

[2] G. Box, G. Jenkins, and G. Reinsel. *Time Series Analysis, Forecasting, and Control, 3rd edition.* Prentice Hall, 1994.

[3] J. Brevik, D. Nurmi, and R. Wolski. Quantifying machine availability in networked and desktop grid systems. In *Proceedings of CCGrid04*, April 2004.

[4] J. Brevik, D. Nurmi, and R. Wolski. Predicting bounds on queuing delay for batch-scheduled parallel machines. In *Proceedings of PPoPP 2006*, March 2006.

[5] J. Brevik, D. Nurmi, and R. Wolski. Predicting bounds on queuing delay in space-shared computing environments. In *Proceedings of IEEE International Symposium on Workload Characterization 2006*, October 2006.

[6] A. Bucur and D. Epema. The performance of processor co-allocation in multicluster systems. In *3rd IEEE/ACM Int'l Symp. on Cluster Computing and the GRID, 2003.*

[7] S.-H. Chiang and M. K. Vernon. *Dynamic vs. Static Quantum-based Processor Allocation.* Springer-Verlag, 1996.

[8] S. Clearwater and S. Kleban. Heavy-tailed distributions in supercomputer jobs. Technical Report SAND2002-2378C, Sandia National Labs, 2002.

[9] M. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5, December 1997.

[10] C. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *International Parallel Processing Symp. – Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[11] A. Downey. Predicting queue times on space-sharing parallel computers. In *Proceedings of the 11th International Parallel Processing Symposium*, April 1997.

[12] A. Downey. Using queue time predictions for processor allocation. In *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, April 1997.

[13] C. Ernemann, V. Hamscher, U. Schwiegelshohn, R. Yahyapour, and A. Streit. On advantages of grid computing for parallel job scheduling. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2002.*

[14] C. Ernemann, V. Hamscher, and R. Yahyapour. Economic scheduling in grid computing. In *Job Scheduling Strategies for Parallel Processing*, 2002.

[15] The evergrid home page – `http://www.evergrid.com/`.

[16] D. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992.

[17] D. G. Feitelson. A survey of scheduling in multiprogrammed parallel systems. In *Research Report RC 19790 (87657), IBM T. J. Watson Research Center*, 1997.

[18] D. G. Feitelson and B. Nitzberg. *Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860.* Springer-Verlag, 1996.

[19] D. G. Feitelson and L. Rudolph. *Parallel Job Scheduling: Issues and Approaches.* Springer-Verlag, 1995.

[20] D. G. Feitelson and L. Rudolph. *Towards Convergence in Job Schedulers for Parallel Supercomputers.* Springer-Verlag, 1996.

[21] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling, a status report. In *Job Scheduling Strategies for Parallel Processing.* Springer-Verlag, 2004.

[22] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1998.

[23] E. Frachtenberg, D. G. Feitelson, J. Fernandez, and F. Petrini. *Parallel Job Scheduling Under Dynamic Workloads*. Springer-Verlag, 2003.

[24] J. Gehring and T. Preiss. Scheduling a metacomputer with uncooperative sub-schedulers. In *Job Scheduling Strategies for Parallel Processing*, 1999.

[25] A. L. Goel. Software reliability models: Assumptions, limitations, and applicability. In *IEEE Trans. Software Engineering*, Dec 1985.

[26] B. Gorda and R. Wolski. Time sharing massively parallel machines. In *Proceedings of International Conference on Parallel Processing (ICPP)*, August 1995.

[27] C. Granger and P. Newbold. *Forecasting Economic Time Series*. Academic Press, 1986.

[28] R. Gupta and C. Chi. Improving instruction cache behavior by reducing cache pollution. *Supercomputing'90. Proceedings of*, 1990.

[29] M. Harchol-Balter. The effect of heavy-tailed job size distributions on computer system design. In *Proceedings of ASA-IMS Conference on Applications of Heavy Tailed Distributions in Economics, Engineering and Statistics*, June 1999.

[30] M. Harcol-Balter and A. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 1997.

[31] T. Heath, P. M. Martin, and T. D. Nguyen. The shape of failure. In *Proceedings of the First Workshop on Evaluating and Architechting System dependabilitY (EASY)*, 2001.

[32] F. Heine, M. Hovestadt, O. Kao, and A. Streit. On the impact of reservations from the grid on planning-based resource management. In *International Workshop on Grid Computing Security and Resource Management*, 2005.

[33] R. K. Iyer and D. J. Rossetti. Effect of system workload on operating system reliabilty: A study on ibm 3081. In *IEEE Trans. Software Engineering*, 1985.

[34] D. Jackson, Q. Snell, and M. Clement. Core algorithms of the maui scheduler. In *7th Workshop on Job Scheduling Strategies for Parallel Processing, 2001.*

[35] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[36] D. N. Jutla and P. Bodorik. Improving applications performance: a memory model and cache architecture. *SIGARCH Comput. Archit. News*, 1997.

[37] J.-C. Laprie. Dependability evaluation of software systems in operation. In *IEEE Trans. Software Engineering*, 1984.

[38] I. Lee, D. Tang, R. K. Iyer, and M. C. Hsueh. Measurement-based evaluation of operating system fault tolerance. In *IEEE Trans. on Reliability*, 1993.

[39] W. Leland and T. Ott. Load-balancing heuristics and process behavior. In *Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS '86)*, 1986.

[40] D. Lifka. *The ANL/IBM SP scheduling system*, volume 949. Springer-Verlag, 1995.

[41] D. Long, A. Muir, and R. Golding. A longitudinal survey of internet host reliability. In *14th Symposium on Reliable Distributed Systems*, 1995.

[42] D. D. E. Long, J. L. Carroll, and C. J. Park. A study of the reliability of internet sites. In *Proceedings of the 10th IEEE Symposium on Reliable Distributed Systems (SRDS91)*, 1991.

[43] J. MacQueen. Some methods for classification and analysis of multivariate observations. 1967.

[44] Maui scheduler home page – `http://www.clusterresources.com/products/maui/`.

[45] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. In *IEEE Trans. Parallel and Distributed Systems*, 2001.

[46] M. Mutka and M. Livny. Profiling workstations' available capacity for remote execution. In *Proceedings of Performance '87: Computer Performance Modelling, Measurement, and Evaluation, 12th IFIP WG 7.3 International Symposium*, December 1987.

[47] D. Nurmi, J. Brevik, and R. Wolski. Modeling machine availability in enterprise and wide-area distributed computing environments. In *Proceedings of Europar 2005*, August 2005.

[48] D. Nurmi, J. Brevik, and R. Wolski. QBETS: Queue bounds estimation from time series. In *Proceedings of 13th Workshop on Job Scheduling Strategies for Parallel Processing (with ICS07)*, June 2007.

[49] D. Nurmi, R. Wolski, and J. Brevik. Model-based checkpoint scheduling for volatile resource environments. In *Proceedings of Cluster 2005*, September 2004.

[50] NWS Batch Queue Pprediction web interface. `http://nws.cs.ucsb.edu/ewiki/nws.php?id=Batch+Queue+Prediction`.

[51] V. Paxon and S. Floyd. Wide area traffic: the failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3), 1995.

[52] V. Paxon and S. Floyd. Why we don't know how to simulate the internet. In *Proceedings of the Winder Communication Conference*, December 1997.

[53] B. Plale, D. Gannon, J. Brotzge, K. Droegemeier, J. Kurose, D. Mclaughlin, R. Wilhelmson, S. Graves, M. Ramamurhty, R. D. Clark, S. Yalda, D. A. Reed, E. Joseph, and V. Chandraeskar. CASA and LEAD: Adaptive Cyberinfrastructure for Real-Time Multiscale Weather Forecasting. *IEEE Computer*, 2006.

[54] J. Plank and W. Elwasif. Experimental assessment of workstation failures and their impact on checkpointing systems. In *28th International Symposium on Fault-Tolerant Computing*, June 1998.

[55] J. Plank and M. Thomason. Processor allocation and checkpoint interval selection in cluster computing systems. *Journal of Parallel and Distributed Computing*, November 2001.

[56] C. Posse. Hierarchical model-based clustering for large datasets. *Journal of Computational and Graphical Statistics*, 2001.

[57] G. Schwartz. Estimating the dimension of a model. In *Ann. of Statistics*, 1979.

[58] L. Smarr and C. E. Catlett. Metacomputing. In *Communications of the ACM*, 1992.

[59] W. Smith, I. Foster, and V. Taylor. Scheduling with advanced reservations. In *Parallel and Distributed Processing Symposium*, 2000.

[60] W. Smith, V. Taylor, and I. 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*, London, UK, 1999. Springer-Verlag.

[61] Q. Snell, M. Clement, D. Jackson, and C. Gregory. The performance impact of advance reservation meta-scheduling. In *6th Workshop on Job Scheduling Strategies for Parallel Processing*, 2000.

[62] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and K. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *In Proc. SIGCOMM (2001)*, 2001.

[63] TeraGrid user portal. `http://portal.teragrid.org`.

[64] N. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Transactions on Computers*, 46(8), August 1997.

[65] The virtual grid application development software (vgrads). `http://vgrads.rice.edu/`.

[66] W. Willinger, M. Taqqu, R. Sherman, and D. Wilson. Self-similarity through high-variability: statistical analysis of ethernet lan traffic at the source level. In *SIGCOMM'95 Conference on Communication Architectures, Protocols, and Applications*, 1995.

[67] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, October 1999.

[68] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Networked windows nt system field failure data analysis. In *Pacific Rim International Symposium on Dependable Computing*, 1999.

[69] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), January 2004.

[70] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, U.C. Berkeley Computer Science Department, April 2001.

[71] S. Z. Zhong. A unified framework for model-based clustering. In *Journal of Machine Learning Research 4*, 2003.