4-2016

User-centric workload analytics: Towards better cluster management

Suhas Raveesh Javagal
Purdue University

Graduate School Form
30 Updated 12/26/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Suhas Raveesh Javagal

Entitled
User-Centric Workload Analytics: Towards Better Cluster Management

For the degree of Master of Science

Is approved by the final examining committee:

Dr. Saurabh Bagchi
Chair

Dr. Jennifer Neville

Dr. Xiangyu Zhang

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Dr. Saurabh Bagchi

Approved by: Dr. William J Gorman, III 4/14/2016

Head of the Departmental Graduate Program Date

USER-CENTRIC WORKLOAD ANALYTICS: TOWARDS BETTER CLUSTER MANAGEMENT

A Thesis

Submitted to the Faculty

of

Purdue University

by

Suhas Raveesh Javagal

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2016

Purdue University

West Lafayette, Indiana

Dedicated to my parents

and

all my sisters for their love, support and inspiration.

ACKNOWLEDGMENTS

This dissertation has been possible because of contribution of many people at Purdue. I thank every one who has made this possible and because of whom I have had a memorable graduate research experience.

I would like to express my gratitude to my advisor Prof. Saurabh Bagchi for giving me this opportunity to work on the research project and the freedom to experiment new ideas and explore new possibilities. His constant support and guidance has helped me recover when I faltered and tackle crisis situations.

My earnest thanks to Subrata Mitra, my colleague whose contributions to this project has been indispensable. His knowledge about distributed systems domain was very useful to get the necessary kick-start for the project. During this project, I have learnt a lot from him starting understanding distributed system behavior to LaTeX tips and tricks.

I would like to thank Prof. Jennifer Neville and Prof. Kihong Park, for taking time to clarify my doubts and questions that arose at various times during the project execution.

My earnest gratitude to Chuck Schwarz and Stephen Harrell, from Information Technology at Purdue (ITaP), for helping us get access to the various data sets in spite of their busy schedules.

I thank Todd Gamblin and Adam Moody, from Lawrence Livermore National laboratory, for giving us the data to perform a comparative study.

I would like to thank Michael Shuey and Dr. Phil Cheeseman, from ITaP for their insightful comments and responses for the questions we posed. I would also like to thank Preston Smith, Dr. Carol Song for providing us the clearance to access and release the data sets.

I am also grateful to the staff at Computer science and Electrical and Computer Engineering department of Purdue University, for their various forms of support during my graduate study.

I am very thankful to my friends, Varun Vasudevan and Yudong Cao, and my colleagues at Dependable Computing Systems Lab (DCSL) for their support and invaluable comments that has contributed for improvement in various dimensions.

Finally, I appreciate the financial support from NSF that funded the project discussed in this dissertation.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Javagal, Suhas Raveesh MS, Purdue University, May 2016. User-Centric Workload Analytics: Towards Better Cluster Management. Major Professor: Saurabh Bagchi.

Effective management of computing clusters and providing high quality customer support is not a trivial task. Due to the rise of community clusters there is an increase in the diversity of workloads and the user demographic. Owing to this and privacy concerns of the user, it is difficult to identify performance issues, reduce resource wastage and understand implicit user demands. In this thesis, we perform in-depth analysis of user behavior, performance issues, resource usage patterns and failures in the workloads collected from a university-wide community cluster, *Conte*, and two clusters maintained by a government laboratory. We also introduce a set of novel analysis techniques that can be used to identify many hidden patterns and diagnose performance issues. Based on our analysis, we provide concrete suggestions for the cluster administrator and present case studies highlighting how such information can be used to proactively solve many user issues, ultimately leading to better quality of service.

1 INTRODUCTION

Large-scale high performance computing (HPC) clusters have become common in academic, industrial, and government for compute-intensive scientific and big-data . These clusters solve problems that would take millennia on personal computers, but managing such large shared resources can be very complex. Managing a large cluster requires administrators to balance requirements form a diverse set of users.

Supercomputers have been used by government laboratories for decades, and in the late 1990's clusters became the predominant architecture for these machines. Typically, they are paid for by national programs and managed and maintained by a central HPC center. Time on the machines is allocated through a competitive proposal process. Users with accepted proposals share the machine and can run jobs until they use up their allocated time. This model has been adopted in majority of the government sponsored research labs (*e.g.,* LLNL, LANL, Oak Ridge, NASA etc. in the US) and industry.

Large, focused organizations can afford to buy centralized resources, but in many universities, sub-organizations (e.g., research groups) are mostly independent in terms of deciding their spending budget. Often, no single research group can maintain its own cluster—the hardware and administrative costs would be too high. Universities have begun to widely adopt a *community cluster* model. In this model, sub-groups buy assets (nodes and other hardware), and these are then assembled and managed by a central IT organization. These clusters have flexible usage policies, such that partners in a community cluster have ready access to the capacity they purchase (much like government cluster allocations), but they can use more resources when other groups' nodes are unused. This allows for opportunistic use for the end users and higher resource utilization for the cluster managers. System administrators take care of security patches, software installation, operating system upgrades, and hardware repair. Centralizing this expertise cuts costs and allows disparate groups to buy large machines with competitive prices for volume purchases. Most importantly, researchers can focus on their research.

Managing large clusters is challenging for many reasons.

Diversity of users. In the community-style clusters the user base is composed of students, professors and research scholars coming from a diverse set of backgrounds such as computer science, biology, physics, linguistics, having skills and understanding of computer systems ranging from novice to experts. Likewise, at government labs, there are many different research scientists, as well as university guests and even student interns. These users execute many different types of parallel applications. Some run a large number of short single-core jobs (such as, parameter sweep for a short simulation), and others who execute long-running jobs using many hundreds of cores (such as constructing the spatial configuration of proteins that is the most stable, that is to have the lowest energy state). Thus the jobs span the spectrum of resource usage. This is a consequence of many organizational sub-units.

Diversity of execution environments. It is challenging to create execution environments that are optimized for this wide diversity of users and applications. The execution environment encompasses the totality of hardware configurations (*e.g.,* number of cores on a node, amount of memory on a node, the network backplane, etc.) and software configurations. Pre-installing applications and associated libraries is the only viable solution as the OS versions of packages are often very out of date, and not usable for the bleeding-edge apps. HPC codes depend on particular versions of libraries, sometimes patch them, require special builds, etc. However, pre-installing implies recurring work for the cluster administrators in updating these software packages and ensuring that dependencies between them are considered during any update process. In reality, most HPC centers are understaffed, and installing software in

Table 1.1.
Key observations, possible implications and recommendations from our analysis.

Observations	**Possible implications**	**Recommendations**	**Ref.**
O1: Some *non-preinstalled* libraries and applications are highly popular.	Users are installing the libraries on their own and risk using a non-optimized or buggy version.	**R1**: Such *hot* applications and libraries should be pre-installed, and on fast storage, to improve user experience and avoid job failures.	Sec. 5.1
O2: Use of 17 libraries almost always lead to job failures due to memory exhaustion.	These libraries might have internal memory bugs.	**R2**: We provide a technique by which source of a failure can be localized.	Sec. 5.2, Fig. 5.3
O3: 63% of the jobs use less than 1% of the requested time.	Long queue time, scheduler cannot perform efficiently.	**R3**: Educate users about queuing and scheduling.	Sec. 6, Sec. 8
O4: 70% of the jobs use less than 50% of the requested memory and a few users used more than requested memory.	Wastage of resources, long queue time, other jobs may suffer.	**R4**: Train users about memory profiling and scheduling. Enforce memory throttling.	Sec. 6, Fig. 6.2
O5: Memory thrashing was found in 20% of the total jobs, which were submitted by 75% of the users.	Extreme slowdown and jobs exceed time limit.	**R5**: Train users about software design and use of more number of nodes to avoid memory thrashing.	Sec. 7
O6: For shared and non-shared environment, memory thrashing behavior is exactly opposite w.r.t *processor per node* value.	Jobs fail or performance degrades in a seemingly arbitrary manner for non-expert users.	**R6**: Training should cover the difference of behavior between shared and non-shared environment.	Fig. 7.6, Fig. 7.7
O7: The top I/O and network resource demands are orders of magnitude higher than the median.	Over provisioning of these resources in the large homogeneous cluster.	**R7**: Better capacity planning can multiple sub-clusters tailored toward these high demand jobs.	Sec. 6.2

all the configurations takes a lot of bandwidth. User support team at TACC@UT Austin has only ∼20–25 people supporting 10k users. LC@LLNL has 2.5k users and a similar size support team, but dedicated mainly to a few mission-critical codes. RCAC@Purdue has ∼20 technical staff supporting multiple clusters and ∼1.2k users and NERSC has 6k users and a smaller support team. Thus, the user support system must embrace more automation and heavily use workload analytics to increase management efficiency. In recent works, Gamblin et al. [1] introduced an automated way to manage complex installations, Agrawal et al. [2] introduced monitoring tool for tracking finer details of a build-system and the environment.

Human factors. Users of large clusters typically create and submit their own job scripts that wrap applications within each script. The scripts allow customization of the execution in a wide variety of ways. Due to the complexity of some scripts and the lack of any standard, parsable template, it is often difficult to identify what are the core applications that are being invoked from a script. From the command line too, the user may specify various parameters, such as the memory and the maximum wall clock requirement of the application. Understandably, with the wide diversity of users, such settings may not be optimal. For example, if the user specifies too small a maximum wall clock time for her script to run and the script reaches that limit, then the job is killed. The system admins may want to monitor sub-optimal configurations, understand the source of inefficiency on a per-application basis and finally work with users to correct them. All this has to be done in a manner that the user feels comfortable with, *i.e.,* does not feel her privacy is being intruded or that her expertise is being questioned. Besides, for this approach to be scalable to the large number of applications and users considering a limited number of system admins, there is a need for significant automation in this aspect.

Here we introduce new ways to analyze the workload data that is being regularly collected in major compute clusters, with the focus on easing systems management in such community clusters. The analysis is meant to give insight into the characteristics of the applications that are run on the clusters and the connection between users, applications and their resource usage patterns. The characteristics that we are particularly interested in are those that are often implicated in performance anomalies or inefficient resource usage. For example, analysis of major page faults in the applications. Our user-centric analysis helps us close the loop by contacting a limited set of users and focus on a limited set of applications for manual debugging. The analysis is done in a scalable manner by use of clustering for applications and users.

Table 1.2.
Summary of workload data analyzed analysed on *Conte*, *Cab* and *Sierra*.

Details	University community cluster	Govt. Lab clusters
Duration	Oct 2014 – Mar 2015	May 2015 – Nov 2015
Total number of jobs	489,971	247,888 and 227,684
Number of unique users	306	374 and 207
No. of unique application behaviors	3,373	-NA-

Getting into the specifics, we analyze the workload traces for 489,971 jobs from one of the largest university-wide community clusters with wide variety of user base and 475,572 jobs from two supercomputing clusters hosted by a government lab working on multiple domains of cutting edge science, as highlighted in Table 1.2. We also make all the workload data from our community cluster available for the researchers through an open repository [3].

Some of our key observations and their implications are as follows — the more detailed list is in Table 1.1.

1. The most visible contrast in the nature of workloads between university and government clusters is evident in the total CPU hours consumed by jobs running at different scales (Fig.1.1). Most of the resources in university cluster are used by small jobs (less than 64 cores) while the government laboratory resource consumption is dominated by very large jobs.

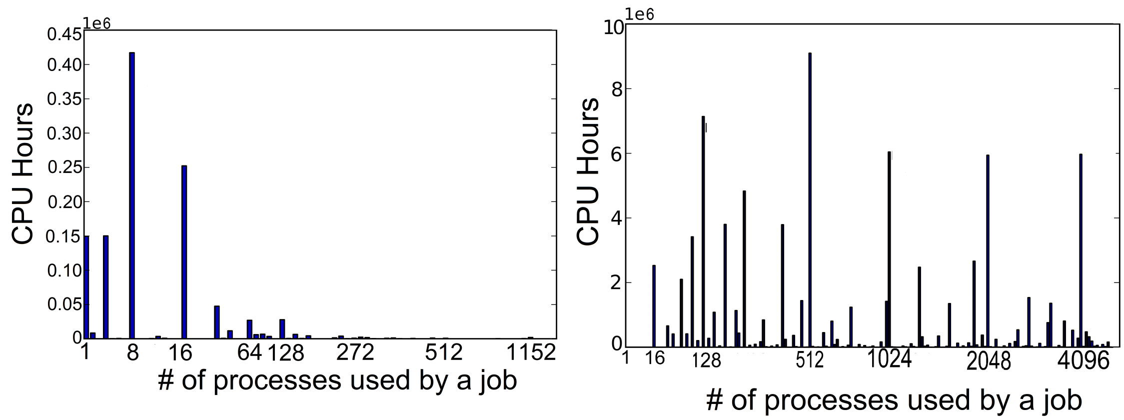

Figure 1.1. CPU hours consumed by jobs of different scales (*Conte* vs *Cab*). Figure on left: *Conte*, Figure on right: *Cab*

2. Some libraries are overwhelmingly popular across applications and users. These can be hand-optimized, replicated and placed in local storage, and pre-installed on these systems. We found that out of 3080 unique application libraries that all jobs use, the top 10 libraries are used by 40% of the users.

3. A few users contribute a disproportionately high number of major page faults. We found that the top 10 users of the system contribute to 50% of the jobs with high page faults. After having brief individual discussions with some of these users, we identified the likely causes of these page faults. In the later sections, we identify what configurations of the applications lead to such performance anomalies, *e.g.,* using large number of cores across machines by an application (as opposed to packing these cores on a few machines).

4. Our analysis sheds light to influence capacity planning and future cluster acquisitions. For example, from the university cluster workload, we found that few applications use extremely high network and I/O (4% and 2% respectively). This implies a need for sub-cluster with high bandwidth backplane and close-by storage. This need is also acknowledged by our partners at University's central

IT organization managing the cluster and our observation will figure in their future acquisition decision.

There have been quite a few works on workload characterization in general [4–6] and failure analysis [7–9], but none of those tried to understand user behavior in order to provide better customer support and improve resource planning and overall quality of cluster management. Moreover, one of our workloads gathered from a community cluster has a unique diversity of users not found in many other type of workloads.

Concretely, this work makes the following contributions:

1. Analysis of usage trends and workloads on large clusters;
2. A novel set of techniques for analyzing cluster workloads:
   (a) Classifying applications without violating user privacy,
   (b) Statistically tracing failure root causes to libraries,
   (c) Predicting reasons for job failures using exit codes and syslog messages, and
   (d) Detecting anomalies in memory usage and other resources;
3. Validation of the utility of our techniques through discussions with users; and
4. A set of recommendations for system managers to *proactively* provide service to users based on the results of our analysis.

2 DETAILS OF CLUSTER HARDWARE

Node hardware, network and filesystem:

University community cluster (Conte): University community cluster has 580 nodes each with two 16 core Intel Xeon E5-2670 processors, two Xeon Phi accelerator cards and 64GB of memory. Nodes are connected through a 40GB/s Infiniband (IB) network. Home directories and pre-compiled application directories are available from each node via NFS and sustain approximately 2GB/s. These filesystems are used university wide and are accessed through the IP over IB egress. The scratch filesystem used by the jobs for this cluster is a Lustre 2.4 installation that can sustain upto 23GB/s and is connected via the above-mentioned IB network. The total capacity is 1.4PB with current utilization being 49%. Every node has a local filesystem too but that is only usable by the operating system and not by user applications.

Government lab clusters (Cab and Sierra): The first government lab cluster (*Cab*) has 1,296 nodes each with a 16 core Intel Xeon E5-2670 processor and 32GB of memory. The second government lab cluster (*Sierra*) has 1,944 nodes each with a 12 core Intel Xeon EP X5660 processor and 24GB of memory. In both clusters, nodes are connected through a QDR Infiniband network.

Our dataset and corresponding analyses are more complete for *Conte* than for the government lab clusters *Cab* and *Sierra*. For any plot, if no designation is given, it is to be understood that the plot pertains to *Conte*; otherwise, we make it explicit in the figure caption.

Software and Scheduling:

University community cluster (Conte): Each node runs RHEL 6.6 OS. The nodes are administrated using the Kickstart installers and Puppet configuration management software. Along with default RHEL libraries, the environment also provides many other important pre-installed libraries and applications which can be used through command: `module load <name>`. The scheduling is done by TORQUE 4, which is an open source implementation of Portable Batch System (PBS) and with Moab as the resources manager. At submission time, each job requests for a certain time duration of execution (a wall clock time), number of nodes and, optionally, amount of memory needed. These are specified through PBS submission scripts. When a job exceeds the specified time limit, it is killed. A job is also killed by an out-of-memory (OOM) killer, a kernel level memory manager, if it exhausts available physical memory *and* swap space. Job scheduling uses a community cluster allocation method in which, some research groups purchase nodes and get *semi-dedicated* access to the nodes that they purchase through their own queue and can also access a far greater number of nodes from the general pool through a shared queue (called standby queue), on demand and opportunistically. For the purchased nodes, research groups get a service level agreement (SLA) of maximum 4 hours wait time for the job at the head of their group-specific queue. When those nodes are not in use, jobs from the standby queue are scheduled on those nodes with a maximum walltime limit of 4 hours. By default, only a single job is scheduled on an entire node giving dedicated access to all the resources. However, sharing can be enabled using a configuration parameter in the job submission scripts. We found it intriguing to know, and discuss in Section 6, why a user would submit a job in a shared environment while they can get dedicated nodes by changing one configuration parameter in the script.

Government lab clusters (Cab and Sierra): In both the clusters, each node runs TOSS 2.2 OS. Moab is used as a workload manager with SLURM as the native scheduler. Nodes do not have a disk hence memory paging is not supported and jobs would be killed when they run out of time or memory. Typically a job is allowed to use maximum of 258 nodes. However under special circumstances, it is possible to use larger number of nodes or even get a dedicated access to the cluster. There are Moab accounts associated with every project and these accounts are assigned a target share of the machine. Every submitted job must specify a Moab account so that resource usage (typically processor minutes) can be tracked and charged to the associated account. The fair-share scheduling scheme is used [10].

3 DATA SOURCES AND COLLECTION METHODOLOGY

Our data collection uses five major components, namely, accounting logs for submitted jobs collected from TORQUE, performance metrics for each node (compute or storage) collected through TACC Stats, `syslog` messages at all the compute nodes, list of shared libraries (*.so) used by the jobs, and job scripts written by the users to submit the job.

Accounting logs: The accounting logs provides job scheduling related details such as the job id, queue name, submission time, start and end timestamps, user's id, requested resources such as: number of nodes, processors per node, walltime limit and memory limit. It also contains on which nodes job was actually run, aggregated resources consumed by the job and an exit status (denoting, whether job ended successfully, crashed, or exited due to configuration error or time limit etc.). A snapshot

```
10/01/2014 00:05:55;E;1660509.machineIP.uni.edu;user=U
group=G jobname=test1 queue=Q ctime=1412131890 qtime=1412131890
etime=1412131890 start=1412136353 end=1412136475
owner=user1@machineIP exec_host=node19/core0/core1/core2/core3
Resource_List.naccesspolicy=shared Resource_List.ncpus=1
Resource_List.neednodes=1:ppn=4 Resource_List.nodect=1
Resource_List.nodes=1:ppn=4 Resource_List.walltime=00:10:00
Exit_status=0 resources_used.cput=00:01:13 resources_used.mem=20mb
resources_used.vmem=25mb resources_used.walltime=00:02:02
```

Figure 3.1. Example of an accounting log for *Conte*

of the raw accounting logs is provided in Fig. 3.1.

TACC Stats: TACC stats [11] data provides more fine-grained resource usage profile on all the nodes used by the job. For each node, TACC Stats data have periodic snapshots of various system metrics which include: usage metrics for the local disk and the Lustre filesystem, Infiniband and IP network traffic, and process and memory statistics.

Syslog: The Syslog data comprises of kernel and system messages of all nodes. It contains memory errors, OOM killer messages, filesystem status, etc.

Library lists: The library list data captures the periodic snapshots of the shared libraries accessed by the jobs in each computing node. Consequently, many such snapshots will be created corresponding to a long running job. Moreover, a job using multiple nodes will have files corresponding to libraries loaded in each node. We aggregate this information for a job and use a novel technique to group the applications based on their libraries used. This is elaborated in Sec. 4.

Job script: A user writes shell scripts to layout the tasks a job would perform. Depending on the expertise of the user, the complexity of a job script varies greatly, from a simple task specified through a single command line to a pipeline of tasks with complex setup for each. However, all scripts must have a set of PBS directives specifying the resource requirements of the job to the scheduler. An example is shown in Fig. 3.2.

For our analysis, we align all the data points using the synchronized time on all the nodes within each cluster.

```
#PBS job submission script
#PBS -q standby # Use standby queue
#PBS -l nodes=8:ppn=16 # Request 8 nodes & 16 cores per node
#PBS -l walltime=04:00:00 # Will timeout and be killed after 4 hrs
#PBS -l pmem=2gb # Needs maximum 2GB physical memory
```

Figure 3.2. Example of resources requested through PBS job submission script

Data collection methodology: The data collection is done in its own distinct way for each types of data. The users' job scripts are taken from the jobs spool directory by a `cron` job and put into a `mysql` database using `pbsacct` [12]. Then specific job scripts indexed by the job-id are pulled from the database. The accounting logs for the specific time periods are used to find the job-ids and extract those specific job scripts. The accounting logs corresponding to each job are collected from TORQUE. The performance statistics are collected by TACC Stats at regular intervals of 10 minutes. Increasing this frequency was found to cause occasional disruptions to running jobs and was therefore not possible. The library lists are collected every 10 mins for each job using the `lsof` tool [13]. `lsof` reports list of shared libraries being used individually for each process of the job. Thoroughly anonymized version of the workload data from *Conte* were made available to the research community for further analysis through an open repository, with the first release being in November 2015 [3].

4 A TECHNIQUE TO CLASSIFY SIMILAR APPLICATIONS

In this section, we discuss how we classified “similar” applications for the purposes of further analyses[1]. Our classification relies on collecting which libraries are being used at runtime by an application and using the library list for classification. A reader may wonder if we could simply extract the application names from the job scripts and cluster the jobs that use the same application together. This simple strategy runs into two main problems. First, the job scripts are often difficult to obtain due to privacy reasons. For example, for *Conte*, we only obtained this after a lengthy negotiation, long after we had obtained other systems-related data and for *Cab* and *Sierra*, we cannot access the job scripts. Second, job scripts are free form and there is no foolproof way to extract the application name for it. Even if one could extract application name, that may not be the granularity at which we want to cluster for resource analysis. For example, an application like `MATLAB` may have completely different (resource usage) behavior depending on which toolbox is being executed.

There have been prior works, which try to classify applications based on their resource usage and communication patterns [4, 14]. But typically, the clustering is done in a more fuzzy manner, *e.g.,*, either creating clusters according to a specific resource usage (such as, memory) or loose clusters that combine a diversity of applications. With our approach, it is possible to map back from the cluster to specific applications and is thus a less fuzzy approach which relies on using the names of libraries gathered through runtime data collection.

[1]Technically, we classify a job where a user script runs one or more jobs on the cluster. A job invokes one or more applications, though typically it is a single application.

Details of the classification technique: We first merge all the libraries corresponding to a job-id, remove the path information, and keep the unique library names. Due to *address space randomization*, the order of such libraries may vary. So we further normalize the list by alphabetically sorting the library names. Then we generate a hash based on this normalized list of library names for each job. We create the hash using just the base name of the libraries and ignore the version number for our purpose as we do not want to distinguish between two instances of an application using two different versions of a library. Once the hash for each job is generated based on the library list, we group together all the jobs which have exact same hash-value. We call each such cluster an *app group*. These app groups are used in Sec 6.2 for analyzing cluster resource usage.

Evaluation of classification technique: To evaluate the accuracy of our technique described in the Section 4 we performed a controlled experiment using 30 popular distinct applications chosen from the domain of scientific computing, image processing, video streaming, and numerical computing. We found, our technique was able to identify distinct applications with an accuracy of 86.7%, *i.e.,* 26 of the 30 applications were classified into their distinct cluster based on the shared libraries used by them. It was interesting to observe, in many cases the presence of only a few libraries are enough to distinguish between two otherwise similar looking library list of two different applications. We discuss further in Sec. 9 the possible shortcoming due to not including static libraries.

5 USE OF PROACTIVE ANALYTICS

In Sec. 5.1 we discuss how to improve customer service by proactively understanding their salient need. In Sec. 5.2 we analyze job failures and in Sec. 5.3 suggest how root-cause of many of these failures can be identified, debugged and fixed using the power of large volumes of data available at cluster administrator's disposal.

5.1 Where to Focus Attention?

Large clusters usually have a diverse user base running various applications which uses a large variations of libraries. However, not all applications or libraries are equally popular. Similarly, there are some users who use the cluster much more frequently than the rest. In order to provide a more effective customer support with limited personnel resources (ratio of IT administrators to users served varies between 1:100 and 1:1000 at large universities and government labs), cluster management must prioritize their efforts. We now discuss how analysis of workloads can help in such decision making.

Hot libraries: When running their applications, users link with various libraries, some of which come with the OS distribution, some are pre-installed[1] in the environment, and rest are downloaded and installed by individual users. In this section we characterize usage patterns of these libraries in *Conte* across all the jobs and all the unique users.

We identify unique libraries from all the jobs and discard the libraries that come with

[1]For example, in *Conte*, using the *module load* command, user can access to pre-installed libraries.

default OS distribution, libraries from */usr/lib64* and /lib64 — these would falsely skew our results.

As a result we got 3,080 unique libraries from which we create sorted histograms by counting how many times each one was used by a job or by an unique user, as shown in Fig. 5.2 and Fig. 5.1 respectively [2]. We also highlight the top 10 most frequently used libraries. Clearly some libraries are used much more often. *e.g.,* in Fig. 5.2, the top-most library is used 4X more frequently than the 50-th most used one. Similarly, in Fig. 5.1, out of 3,080 libraries, each of the top 10 libraries are used by approximately 40% of the users. We recommend, this information about frequently invoked libraries can be used to implement better software caching mechanisms which can improve performance. For example, optimized versions of these libraries for the specific execution environment can be pre-cached in memory or installed on SSDs for faster access. This will not only relieve many users from going through a complex installation process, it will also minimize the risk of using a buggy or unoptimized version leading to performance problems.

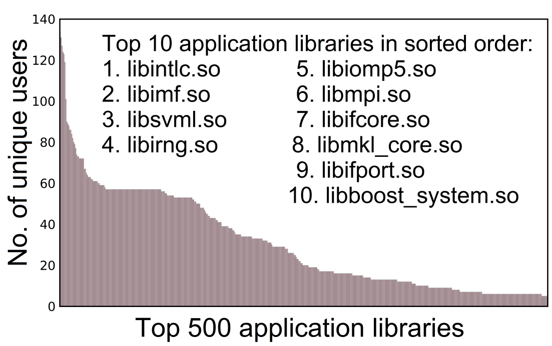

Figure 5.1. Histogram of top 500 libraries used by unique users *Conte*

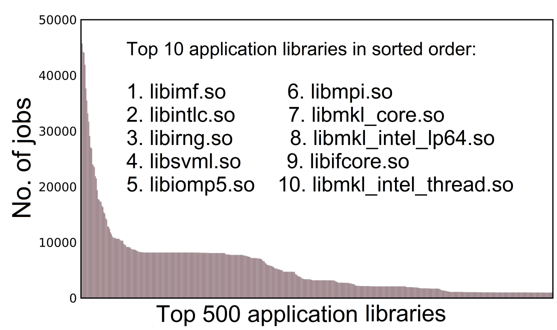

Figure 5.2. Histogram of top 500 libraries used by the jobs in *Conte*.

[2]For subsequent figure captions, if no cluster is explicitly mentioned, it is to be understood that the result pertains to *Conte*.

Reality vs. speculation: We did a reality check to see what percentage of these popular libraries identified by our analysis were actually pre-installed on *Conte*. We first loaded all the available 117 unique *modules* (*i.e.,* applications) in the environment and extracted 950 unique pre-installed libraries from all the paths appended to `LD_LIBRARY_PATH`. We found only 10 out of top 50 and 188 out of top 500 most popular libraries were pre-installed. This is probably because the currently pre-installed libraries were chosen based on intuitions and a quantitative analysis such as ours can inform library installation decisions.

Hot applications: We also tried to identify what applications are most commonly used, among those which are pre-installed in the environment. In Table 5.1 [15], we show the top 10 most heavily used out of 117 pre-installed applications in *Conte*. By analyzing job script where users explicitly *load* these modules, we discovered that many of these pre-installed applications were rarely used and can be replaced by other applications identified by our suggested analysis technique. Fig. 7.5 shows the sorted

Table 5.1.
Top 10 applications used in *Conte*. This is obtained by analysing the user scripts and extracting applications used via *module load*.

R	Devel	Matlab	NEMO5	LAMMPS	NCL	SRA-Toolkit	Gromacs	BWA	TopHat

histogram of jobs submitted by top 50 unique users in *Conte*. It is evident, first few users contribute most of the jobs. These *premium* users should enjoy increased priority from the customer support while resolving any issues. Surprisingly, we found the *top most user* also suffers from significant major page faults (we will discuss this in detail in Sec. 7), highlighting the need for proactive customer support system.

5.2 Why Jobs Fail?

Failed jobs translate into wastage of resources in a cluster and therefore understanding why jobs fail in a cluster is important for the cluster management team as they can proactively use this information to either provide support to the user if the problem is at the user's end or identify any problems in the cluster's hardware or software stack. Each submitted job completes with an exit code which can interpreted to get an indication of the reason for a job's failure. Usually these exit codes are set by the job scheduling system or by the kernel but users can also set their own exit status. However after analyzing job scripts and discussing with cluster administrators of various supercomputing centers, we identified a general exit code convention as follows. Exit code 0 denotes a successful run. Negative error codes usually indicate a failure of the scheduler or the nodes. In the absence of user specified code, exit code from the last executed command in the job script is reported. Exit code $\geq$ 128 or $\geq$ 256 can be decomposed as `128 (or 256) + a system signal` where the system signal can be of various kinds such as SIGTERM/SIGKILL (memory exhaustion), SIGSEGV (segment violation), SIGBUS (file system error), SIGILL/SIGFPE (bad operation), etc., indicating the root case of an unsuccessful job termination. However, in TORQUE, exit code 1 indicates a generic error and cannot be classified to any particular category.

What exit codes do users usually provide? We further analyze what are the exit codes that are usually provided by the users, so that we can disambiguate and interpret the final job exit codes. We first identified all the 84,163 *unique* job scripts submitted to *Conte* using hashing. We found 22,665 of those unique scripts had user specified exit code for error handling. Further, we found that the most common user provided error code is 1 (26%). Users also used various positive exit codes in the range [2-99] to specify error conditions. We found only 5 explicitly specified negative exit codes [-1,-2,-3,-20,-30]. 44 scripts explicitly used exit code 0 at the end to mark success.

Classification of failures with the help of exit codes: An analysis of the job's exit codes revealed in *Conte* 16.2%, in *Cab* 4.4% and in *Sierra* 3.8% of the jobs had failed (exit code not equal to 0). The lower failure rate in *Cab* and *Sierra* may be due to lower resource usage, more experienced users, or better management. After analyzing exit codes along with other information such as the job runtime and resource usage information, we classify the failed jobs along with the reason that resulted in the job failure[3] in Table 5.2. We found in *Conte*, about a fifth of the *failed jobs* were killed due to exceeding the requested wall clock time and the next most significant factor was due to memory getting exhausted[4]. Surprisingly, we found in all 3 clusters, time limits are often not very strictly enforced and we found many instances where jobs were not killed even after exceeding requested times. This is also reflected in the statistic in Table 5.2 where number jobs failed due to timeout is much less than the number of jobs exceeded the time limit (Fig. 6.9 and Fig. 6.10). In Sec. 9 we further discuss the validity of these interpretations by analyzing syslog messages.

Table 5.2.
Few major types of job failures (Time expired, memory, Seg. fault, File path problem, etc.) showing % of *failed jobs*.

Reason	% of *failed* jobs in *Conte*	% of *failed* jobs in *Cab*	% of *failed* jobs in *Sierra*
Time expired (timeout)	20.3	59.8	63.2
Memory exhaustion	15.2	5.6	6.4
Segmentation Fault	9.3	4.0	1.9
Quit/keyboard interrupt	5.3	5.8	10.1
File system/path problem	3.7	6.7	3
Self abort/assert failure	0.6	1.3	1.5

[3]These are not root causes, rather the manifestation of the problem

[4]Failure reasons for many jobs (upto 6.2%in *Conte*) could not be classified as they exited with generic code 1.

5.3 Whom to Blame: Finding *Buginess* of a Library

Almost all the jobs executed on a cluster use a host of libraries, some of which are third party, some are common libraries provided by the environment, and some are written by the users. If certain kind of job failure occurs significant number of times and affects multiple users, it is worthwhile to consider if such a failure is caused by a particular library or due to interaction from a set of libraries. We now illustrate how the source of a certain kind of failure can be narrowed down to few libraries using statistical analysis. We introduce a metric called *buginess of a library* as follows.

Definition: *Buginess* of a library L for a given job failure type F denotes the likelihood that a job failed as F due to a bug in L.

Buginess can be expressed as a value $\in [0, 1]$ according to our proposed Bayesian formula:

$$Buginess\ of\ L|F = \frac{P_r(L|F)}{P_r(L|F)+P_r(L|job\ succeeded)}$$

Where $P_r(L|F)$ and $P_r(L|job\ succeeded)$ denotes the probability that a job using library L failed with type F and succeeded respectively. It should be noted that $P_r(L|F)$ and $P_r(L|job\ succeeded)$ are not complementary to each other because library L might be also be associated with *other* type of failures. There are two reasons why buginess value should be tied to a particular type of failure. First, a library can have different strength of association with different types of failures and most likely cluster administrators might only be interested in few of these types while ignoring others (*e.g.,* killed by user, config error). Thus a general buginess value does not help. Second, a detailed analysis for each failure type calls for different diagnosis methods. For example, for time expired (also referred as timeout) failures, detailed analysis must profile time taken by each function call to that library whereas debugging for memory errors are more difficult requiring guard codes, hardware watch points and memory leak detection mechanism.

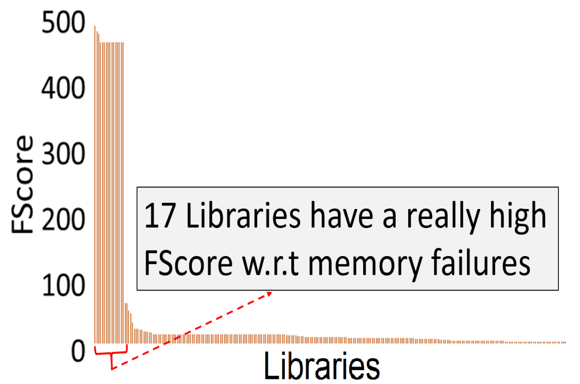

Figure 5.3. Distribution of *FScore* values w.r.t failures due to memory problems in *Conte*.

How to interpret a buginess value ? Buginess value considers both successful jobs and failed jobs associated with a library. If a library was never associated with a failed job, its buginess value is 0. On the other hand, if that library was always associated with a failed job and never used by any jobs that succeeded, its buginess value is 1. Thus, libraries with buginess value > 0.5 should be suspicious and the ones with a value closer to 1 should be carefully inspected by the cluster administrator.

Identify most suspicious libraries: However, buginess value alone is not good enough. For example, in Fig. 5.3, both L_1 and L_2 have same buginess value (*i.e.,* 1), but L_1 is more likely to contain the bug leading to the failure as it appeared more frequently in failed jobs. A naive frequency analysis would not help either as it would disregard how a library was also associated with successful jobs. For example, even though L_2 and L_3 both appeared twice within failed jobs, L_3 was also associated with a successful job and hence should be less suspicious.

Hence, we calculate an *FScore (final score)* by multiplying the buginess value by the frequency of a library corresponding to a type of failure as outlined in Algorithm 1. Suspicious libraries can be identified for detailed investigation by choosing the top K, ordered by the *FScore*.

Table 5.3.
Example: L_1 is more suspicious than L_2 as it affects more jobs eventhough they have same buginess value. L_3 has lower buginess than L_2 as L_3 is associated with a sucessful job.

Libraries used	Job failed ?
L_1,L_2,L_3,L_4	Yes
L_1,L_2,L_3,L_5	Yes
L_1,L_5,L_6	Yes
L_1,L_7,L_8	Yes
L_3,L_4,L_9	No

Algorithm 1 Identify top K most suspicious libraries for detailed analysis

Input: *LibList* ← List of libraries

Input: F ← The failure type being investigated

Input: *JobList* ← List of all jobs with corresponding failure types and used libraries

FScore ← []
for all L in *LibList* **do**
    buginess ← calculateBuginess(L, F, *JobList*)
    failFrequency ← getLibraryFrequency(L,F,*JobList*)
    FScore[L] ← *buginess* * *failFrequency*
end for
sortedLibs ← sortLibrariesBasedOnFScore(*LibList*,*FScore*)
return getTopK(*sortedLibs*)

Fig. 5.3 shows the FScore distribution for jobs failed due to memory problems in *Conte*. As can be seen, only a very small set of 17 libraries can be isolated with really high FScores (they also had buginess > 0.95). We suggested to the admins of *Conte* to perform an in-depth analysis of these 17 libraries by *re-executing* jobs using a memory profiling tool such Valgrind. Our current scoring formula has one disadvantage that it may miss out some of the rarely used yet buggy libraries because we multiply the buginess value by the number of times that library was associated with a failed job. For example, jobs using *libMCEAucdDatabase.so* always failed (buginess =1.0) due to a memory error. Since number of such jobs were really small, the FScore for this library was low and as a result, this library did not show up in our final list of 17. However, this is merely a policy decision for ranking the libraries, as we wanted focus on the frequently used libraries which might have a bug. Different scoring functions can be used to explore other type of ranking.

6 WHAT ARE THE PATTERNS OF RESOURCE REQUEST?

In the job submission script, users can request the number of nodes, number of cores per node (`ppn`), a time duration the job needs to run, and the maximum amount of memory that any process of the job would use, as shown in Fig. 3.2. Generally, these limits are chosen based on the users' intuition or some prediction. In this section we study, how good are the intuitions (or predictions), based on which the users specify these limits. And later, we discuss the consequences of really bad prediction for parameters like maximum runtime and memory. For our analysis involving actual runtimes, we filtered out the jobs that run for less than 30 seconds because many jobs fail early due to configuration errors and thus should not contribute towards general behavior [4].

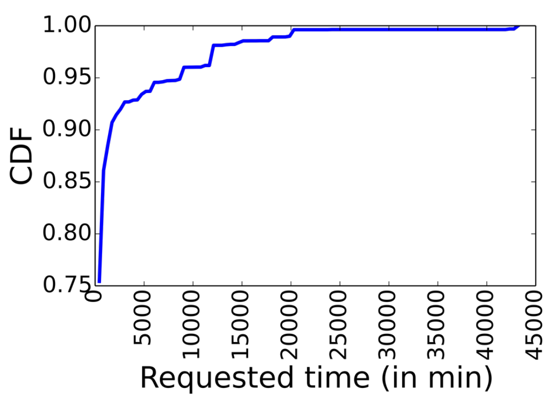

Figure 6.1. CDF of requested walltime

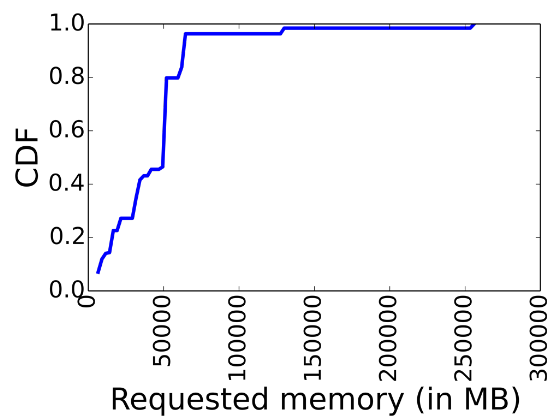

Figure 6.2. CDF of requested memory.

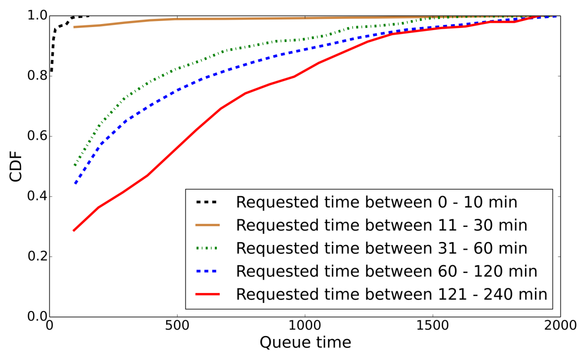

Figure 6.3. CDF of queue time for various requested walltime ranges in *Conte*. As requested time increases, probability of waiting longer in the queue increases.

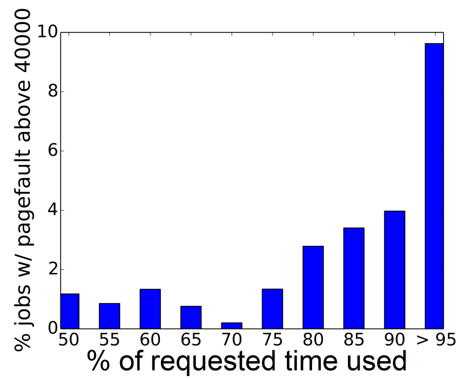

Figure 6.4. Major page faults is correlated with percentage of requested time used by a job (beyond 75% of requested time).

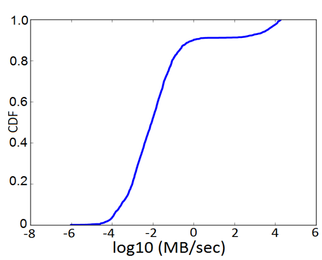

Figure 6.5. Infiniband read rate for app groups.

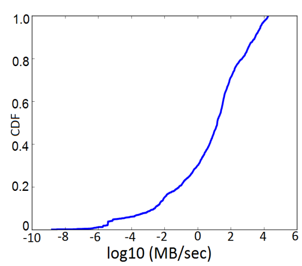

Figure 6.6. Lustre read rate for app groups.

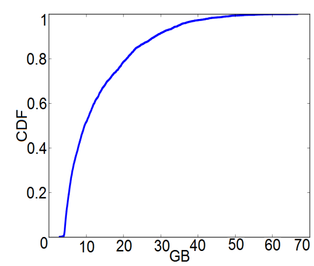

Figure 6.7. Avg. memory used by app groups.

6.1 Runtimes of Jobs

Fig. 6.1 shows the CDF of requested runtime (`walltime` for *Conte*. We found 80% of the jobs were submitted with a walltime limit of 10 hours or less. However, 5 users always requested for extremely long runtime (upto 720 hrs). We contacted them to understand what prompted them to specify such long runtime limits and found their jobs were hanging due to memory thrashing as detailed in Sec. 8.

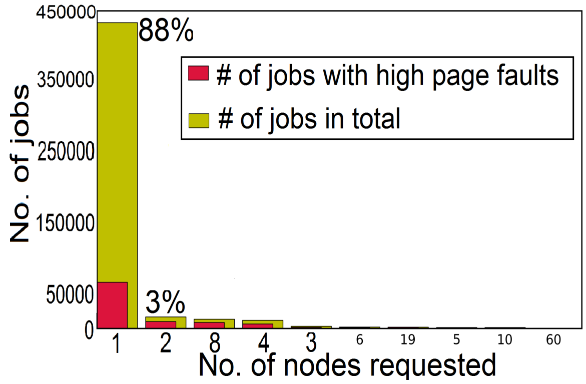

Figure 6.8. Sorted histogram of jobs w.r.t number of nodes requested in *Conte*

Wastage of resources: We further investigated what percent of requested time are actually used by the jobs. To our surprise, we found almost 45% of jobs actually used less than 10% of the requested times in *Conte* (Fig. 6.9) and 15% used less than just 1% in *Sierra* (Fig. 6.10).

From our experience, in most of the large production clusters, when the cluster is busy, the job scheduler gives higher priority to the jobs which request shorter runtime. Hence requesting for a long runtime would mean a long wait in the queue. This is not only frustrating for users but also leads to sub-optimal allocation by the scheduler of jobs to nodes. To validate our argument, in Fig. 6.3 we show how the probability of longer wait in a queue increases with the requested runtime limit of the job in a shared queue in *Conte*. We see that until a requested runtime of 30 minutes there is no significant effect, but beyond that queue time increases significantly. Of course, if certain groups have a dedicated queue and have dedicated resources (in *Conte*) or a high priority account (in *Cab* and *Sierra*) then the behavior will be different. Also we found that around 5.7% of the jobs exceeded the requested time and as a result 3.3% (according to exit code based analysis in Sec. 5.2) were ultimately killed. This again highlights the wasteful impact of poor runtime prediction by the user.

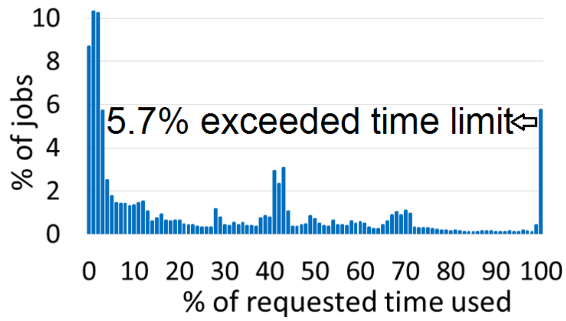

Figure 6.9. Percentage of the requested time used by the jobs. ≈6% jobs terminate due to exceeding time limit.

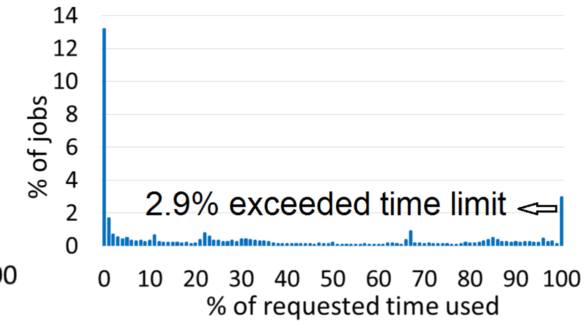

Figure 6.10. Percentage of the requested time used by jobs in *Sierra*. 15% jobs used less than 1% of the requested time and 2.9% exceeded time limit.

We conclude that overestimation of runtime during job submission is a serious problem and severely affects resource usage in both the university cluster and the government lab clusters.

Why do jobs run out of time? As many of the users come from different domains, not everyone is well-versed in scheduling and managing their jobs efficiently in a supercomputing cluster. Hence it is understandable that memory related issues are frequently faced by users. Here, we characterize what percentage of the jobs that run out of their requested walltime experienced massive memory thrashing in *Conte*, most likely resulting in slowdown or exceeding the allotted time limit. In Fig. 6.4, we plot percentage of jobs experiencing severe page faults (peak page fault was more than 40,000 in any 10 minute interval) against the percentage of requested time actually used. We find that the percentage of jobs facing such massive major page faults which used up $> 95\%$ of requested time is almost *5X* higher than jobs which used only $\leq 75\%$ of the requested time. Thus, statistically, major page fault is highly correlated with percentage of requested time used, for the jobs which exceed the 75% mark.

Memory limit: In *Conte*, user can provide a maximum memory limit (`pmem`) in PBS script. We found only `14.8%` of the jobs specified it. As shown in Fig. 6.2 CDF, almost 70% of the jobs ultimately use less than 50% of the requested peak memory. On the other end, we identified 0.6% of the jobs violated the requested maximum memory limit as memory throttling is not enforced in this cluster. Further analysis revealed that these jobs were submitted by only 7 users. We reported this to the cluster staff but since the percentage is low enough not to cause any major disruption we decided not to follow it up with the users. Neither *Cab* nor *Sierra* allows users to specify a maximum memory limit.

Number of nodes and cores: Fig. 6.8 summarizes the request pattern of the jobs for number of nodes in *Conte*. 80% of the jobs request for only 1 node while 2, 4 and 8 are also popular choices for number of nodes. Very few jobs requested upto 256 nodes. One explanation can be: the average number of nodes owned by research groups is 9 and users like to stay within their allocation.

Similarly, in Fig. 7.4, we plot percentage of jobs w.r.t total number of cores requested in *Conte*. Total number of cores is number of nodes multiplied by `ppn`. Around 40% of the jobs are run on single core. We hypothesize, a large fraction of jobs in *Conte* corresponds to non-parallel applications and small scale test jobs — typically used for functionality testing before submitting larger scale parallel applications.

CPU-Hours is a better way to understand resource usage pattern: The distribution of CPU-Hours used by a job on a cluster often gives a better view of resource usage pattern than the size of the job in terms of the number of processes. For example, in Fig 1.1, for both *Conte* and *Cab*, we show CPU-hour distribution of the jobs of different scales (*i.e.,* how many process they use). As expected, *Cab* being a high churn national lab, a large fraction of resources are used by large scale jobs (with processes 512, 2048, 4096, etc.). However, a mere counting of number of jobs would show only one node and 16 cores are the dominant request patterns, which, as explained before, is skewed by the large number of small-sized jobs.

Sharing of resources: By default, in *Conte* a node is not shared between jobs. However, a user can enable sharing by setting `naccesspolicy=shared` in her job script. After analyzing the accounting logs, we found 23.6% jobs had the shared setting. We contacted the users to know why would one prefer to share the nodes when they can get dedicated access to a node by default. We found the following are the key reasons behind that decision: a) users often perceive that a partial node job will start faster than a whole node one - and within certain limits this is true, b) some research groups own limited number of nodes and try to maximize the usage within the limitation of their assets before requesting from the general pool, c) legacy scripts which had that configuration, and d) good old altruism.

6.2 Is the Current Resource Provisioning Good Enough?

For effective cluster management, it is also necessary to understand whether other resources such as network, filesystem, and memory are being utilized well. Trends for such resources can also help in improving the provisioning strategy or future acquisition decisions. In this section we present Infiniband, Lustre, memory usage patterns across the app groups in *Conte*. We explore how the various resources of the system, specifically, Lustre file system and Infiniband network are used by the jobs. The data necessary for this analysis is extracted from the TACC Stats for each jobs. As the TACC Stats provides information at intervals of every 10 mins we extract the values and calculate the differences between consecutive elements and obtain the peak among the difference. Further we obtain corresponding rate (divide it by 600 seconds). Using this peak rate for each metric for all the jobs we plot a cumulative distribution function (CDF) for each metric. The results and the respective observations are provided in the respective sections.

Infiniband (Network) usage: For *Conte*, we calculated the average of the peak Infiniband data rates experienced by jobs in the same app group[1]. The CDF is in Fig. 6.5. The Infiniband data rate includes the communication between different processes of a job. We see a stark contrast in the resource demand across different app groups. The plot is split to emphasize the order of magnitude difference in IO usage across different app groups. Almost 90% of the app groups use less than 0.8MBps while a few app groups go upto 20GBps. Such information regarding the huge difference in Infiniband usage can be used during capacity planning for future clusters. Clearly, scheduling both types of jobs on the same cluster would mean under-utilization of the resources. Instead of over-provisioning all the networks, there can be a virtual sub-cluster with higher performance characteristics. The cluster nodes connected through a high-end interconnect can be kept exclusively for jobs with such extreme resource demands. In fact, for the next acquisition cycle at our university (after our analysis period), precisely such a decision was made driven in part by the results of the analysis and a separate cluster is provisioned for high resource demand applications. A cursory examination of the jobs running on that cluster indicate a preponderance of bioinformatics applications which cause high memory usage and inter-process communication.

Lustre (IO) usage: For *Conte*, we extract IO usage from the Lustre filesystem metrics present in TACC Stats and plot the CDF of average peak data rate for jobs of same app group. We choose the peak value per job to understand maximum resource demands for the jobs. Similar to Infiniband usage, we found 90% of the app groups' IO usage is less than 1GBps and a few app groups use IO at a rate close to the rated 20GBps. Thus it is possible to partition the cluster into sub-clusters with different IO performance specs.

Memory usage: In the same spirit, we calculate average memory usage by the jobs belonging to the same app group in *Conte*. The CDF is in Fig. 6.7. We see that

[1]Peak usage is calculated across all the processes of a job and all monitoring intervals. Then an average was taken over all the jobs in the app group.

unlike Infiniband or Lustre usage, memory usage by different app groups does not show any large discontinuity. Thus, scheduling jobs on different partitions of a cluster with available physical memories would not be substantially useful for improving the memory utilization.

Strong scaling vs weak scaling intent: Certain cluster usage patterns are not obvious from the accounting logs or from the raw numbers of resource usages. For example, strong scaling and weak scaling are two standard ways in which scientific applications are usually studied to suit different purposes. In a strong scaling run, user would run the application with more number of nodes while keeping the problem size the same. Thus resource usage per node would decrease proportionally. For a weak scaling run, user would use more number of nodes while increasing the problem size as well. However, looking at the system logs, there is no easy way to know if certain applications are being investigated for strong or weak scaling. We wanted to find out how such statistics can be indirectly inferred from the existing workload data that we had already gathered. The cluster administrators were also keen to know such results as it helps them in better cluster management. For example, if a user is using weak scaling and if resource demands per node were high at small scales, then they would continue to be high at the larger scales and the user may be better advised to switch to a higher-provisioned cluster. Since it is difficult to understand the input size from the script, we further assume, IO usage, *i.e.,* the amount of data read from the Lustre filesystem, is a good indicator of input size. We found that only 71 app groups, out of 3,373 (2.1%), had enough data to infer statistically valid scaling trends, *i.e.,* there were application runs on more than 3 different scales and also had at least 100 job data points at each scale. We found, out of the 71 app groups, 14 of those app groups were subjected to strong scaling (19.7%), 26 were subjected to weak scaling (36.7%) and for the rest, neither trends was observed.

7 ANALYSIS OF MEMORY USAGE: THE ETERNAL PAIN

Memory usage related issues are in general a big reason for unsatisfactory application performance. The most severe impact of memory issues shows up as memory thrashing, which is when applications exhaust the available physical memory on the node and start swapping pages from the disk (also called *major page faults*). On *Conte*, we have seen that when an application experiences memory thrashing, its runtime may increase from 10's of minutes to long hours. At the extreme, after exhausting even the swap space, the application would be killed by an out-of-memory (OOM) killer to save the node from crashing. In this section, we characterize the impact of memory thrashing by analyzing the workloads from *Conte* [1]. TACC stats periodically reports snapshots of the number of major page faults experienced by a job in the last 10 minutes time window, based on which we classify thrashing behavior into two types:
a) *Chronic thrashing:* The job experiences major page faults throughout its run, but the absolute numbers per window of reporting are not significant enough to raise any alarm. We ignore this type of thrashing for our analysis.
b) *Severe thrashing:* The number of major faults within the time window goes beyond a certain threshold indicating a severe memory thrashing experienced by the job. Our analysis targets only these type of thrashing cases.
Since an application may experience high page faults initially, due to the overhead during start up, we exclude first 10 minutes of its run-time from our analysis.

We define ***peak major page fault*** of a job as the maximum page faults experienced by a job within any reporting time interval across all the nodes on which the job was running. In general, such maximum numbers are noisy, but for our analysis we

[1] There are no disks in *Cab* and *Sierra*, hence no paging

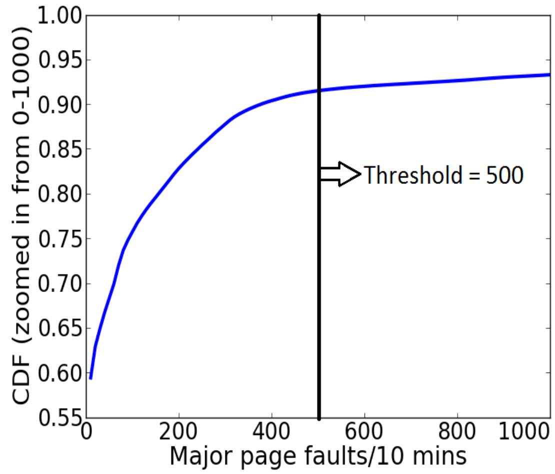

Figure 7.1. CDF of peak major page fault of all jobs, threshold set at 90^{th} percent value

wanted to be conservative in identifying such cases as we did not have the ground truth. We first identify a threshold for peak major page fault by calculating the CDF across all the jobs (Fig. 7.1). It can be seen that 90% of the jobs experience less than 500 major page faults within any 10-minute time window, therefore we chose 500 major page fault per 10 min time window as our threshold and use this for all subsequent analysis.

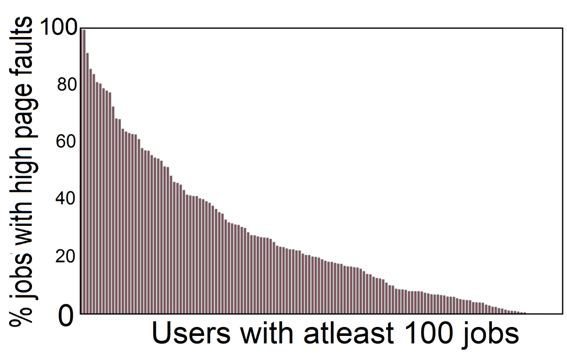

Figure 7.2. Percentage numbers of user's jobs crossing the threshold peak major page fault in *Conte*

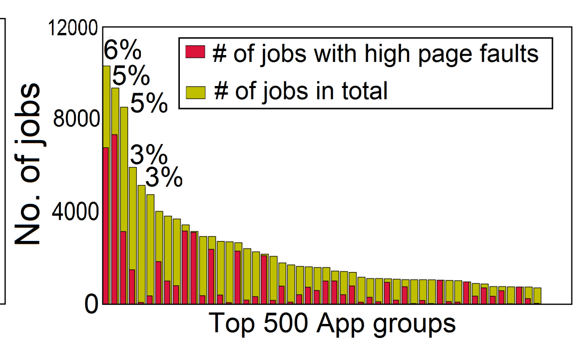

Figure 7.3. Sorted histogram of app groups w.r.t number of jobs crossing peak major page fault threshold in *Conte*.

Fig 7.5 show what percentage of jobs submitted by top 50 most active users suffered from severe page faults (*i.e.,* peak page fault was more than the computed threshold value). The yellow bar denotes all the jobs submitted by the user and the red bar denotes the jobs that suffered severe major page faults. While for most of them it was not very significant, certainly the jobs belonging to the top user were affected significantly. Further in Fig. 7.2 we show the histogram of jobs for unique users sorted based on the percentage of jobs that suffered major page fault. For this analysis, we filter out users who have submitted less than 100 jobs. Clearly, most of the jobs from top few users suffered from major page faults and they definitely need assistance from cluster support staff. In fact, we contacted few such users to understand the issues. This is discussed in Sec 8.

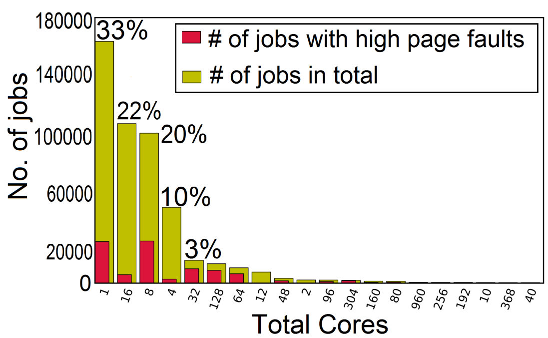

Figure 7.4. Sorted histogram of jobs w.r.t number of cores requested in *Conte*

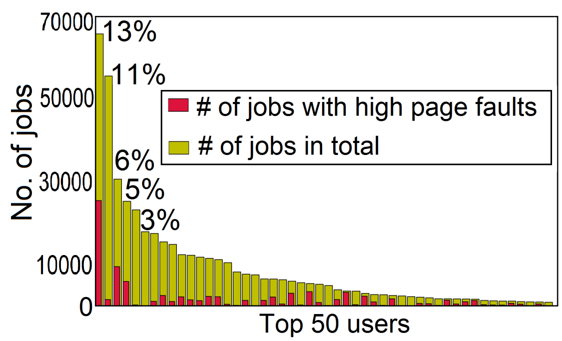

Figure 7.5. Top 50 users sorted based on number of jobs submitted in *Conte*

We also analyzed what percentage of the jobs in the most popular app groups (sorted based on number of jobs) suffer from major page faults. This is presented in Fig. 7.3. To our surprise, we found that top two app-groups suffer from major page fault quite severely. To provide a better quality of service, the cluster administrators should understand why these applications suffer from memory exhaustion and educate the users who use these apps about techniques to reduce the memory pressure. The support staff might also pre-install a properly optimized and memory leak free version

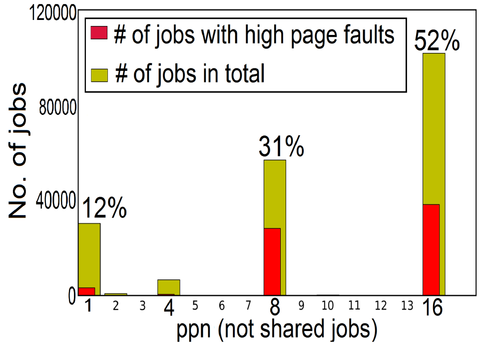

Figure 7.6. Non-shared env: effect of ppn on major page faults in *Conte*

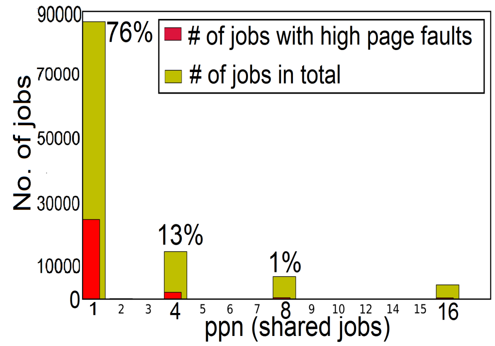

Figure 7.7. Shared env: effect of ppn on major page faults in *Conte*

of these applications, if not already available. In Fig. 6.8 and 7.4, we show the relation between the percentage of major page faults with the number of cores and the number of nodes that a job runs on.

Effect of process placement on memory issues: As discussed earlier, user can set the number of cores per node to be used through the parameter `ppn` in the PBS job script. Since each node has 16 cores in *Conte*, `ppn` can take a value from 1 to 16. Besides, the user can also specify if the job can be run in a shared environment where the remaining cores of the nodes will be used by other jobs. We analyzed how the choice of `ppn` affects the memory usage issues for both non-shared and shared environments as summarized in Fig. 7.6 and 7.7. The observations are quite fascinating as we found that the value of `ppn` affects major page fault in exactly *opposite* directions between non-shared and shared environment. In a non-shared environment, as `ppn` increases, the page fault increases. This happens because the more processes on a node imposes greater memory pressure on the node. On the other hand, in a shared environment, as `ppn` increases, the page fault decreases. This happens because the level of interference from other co-located applications decreases. There are some other production clusters where in shared mode, the memory allocated to a job is made proportional to the requested `ppn` and memory throttling is used based on that. It would be an interesting future-work to analyze whether workloads on those clusters would also exhibit similar behavior.

8 CASE STUDIES

We now present a few case studies through which we verified that our analysis accurately identified the users who were suffering from performance problems or wasting cluster resources. We also summarize the cause of the problem and the remedies suggested by us or cluster support staff.

On the university community cluster:

We contacted the top 10 users of *Conte* who were experiencing severe memory thrashing as identified by Fig. 7.2. Eight of them responded, but one of them did not share the details due to privacy reasons. Another user claimed that he saw no performance degradation; we hypothesize, that the user may not have observed a *change* in performance, or the application was suffering all along. The rest of the users confirmed a significant performance slowdown with their jobs and through our interactions, we unearthed the root causes. Our interactions involved interviewing the users, review of job scripts, and in few cases, analyzing the application code.

Too many processes per node: Three users faced similar problems which arose because of using too many processes per node, leading to exhaustion of physical memory and increase in page faults. For example, one inexperienced user started seeing out-of-memory errors when he moved from a third party application to his own code, mimicking the same functionality but at a larger scale. He did increase the number of processes but could not increase number of nodes as he was using his group's dedicated queues. The issue was alleviated when we suggested him to use shared queue and decrease the `ppn` but increase the number of nodes.

File append: User started experiencing extreme slowdown (run-time of ∼10hrs instead of ∼40mins) when she modified a self-written bioinformatics python application. Instead of writing to a large number of output files, she was appending to one existing file. She was running many instances of a serial application in parallel, each one operating on certain parts of the input data. Our analysis also detected a huge incidence of major page faults corresponding to her jobs, during the same time period. After debugging the code, we noticed the use of file seek in a loop. Ultimately, we found that the combined effect of appending to a growing file and file seek to arbitrary locations in a loop caused a manyfold increase in memory pressure.

Effect of precision: While running a nano-electronic modeling tool the user experienced extremely long runtime and frequent crashes due to out-of-memory. After some analysis, it was found out that a parameter, `coulomb-cutoff-radius` which controls the precision by limiting the radius of calculation, was set to a very high value leading to high memory consumption. There are two ways to resolve this issue, either to decrease `ppn` or to reduce the radius of the calculation.

Unoptimized MPI communication: A *Weather Forecasting* application was hanging and ultimately exceeding its time limit almost 15% of the time. Our analysis also detected huge page faults corresponding to those jobs. However, this user proactively contacted cluster support staff, who after analyzing the jobs, suggested the use of `I_MPI_DAPL_UD` directive for Intel MPI library. This ensures a many-to-one connection (instead of the standard one-to-one connection) between the MPI processes which reduces the memory consumption. After making this change, the user reported that only 1% of her jobs experienced memory thrashing.

On the government lab cluster:

Since *Cab* and *Sierra* belong to a privacy-sensitive national lab, we only had access to anonymized user data which prohibited us from making direct contacts with the users. However, we still identified a particular user ID (anonyimized) in *Sierra* whose jobs were using only a small fraction of the requested walltime. When we first calculated the distribution showing percentage of requested time used for *Sierra* (as discussed in Sec. 6), we found a staggering 63% jobs used less than 1% of the requested time, which we found suspicious. Further analysis revealed the statistic was skewed by a particular user who runs a lot of tiny jobs which are likely part of larger computational studies. We found that this user accounts for only 5K CPU-hours despite having the largest number of jobs on that cluster. After a detailed investigation, the cluster administrator found that indeed this user was doing parameter studies of solvers but had vastly overestimated the running time of each individual job. More discussions revealed that jobs related to parameter studies and uncertainty quantification in general tend to have such problems and schedulers should be more intelligent while dealing with such jobs. This is an exemplar of the kind of insight that comes out of customer centric data analytics.

9 THREATS OF VALIDITY

We now discuss the validity of the assumptions we made in our analysis, highlight the weaknesses, if any and suggest future improvements.

General applicability of library based classification technique: Proposed library usage based application classification technique (Sec. 4) currently depends on the use of shared libraries as *lsof* cannot collect static library usage information. However, as discussed in Sec. 4, existence of only few distinctive libraries are enough for the classification. For a further reality check, we calculated the *DScore(distinctive score)* of a library. To calculate this, we identify all the *unique* job scripts by hashing and treat them as representing a set of unique jobs (J). Let the size of set J be N. We merge the libraries used by all jobs in J and discard the default OS libraries to get the set S_{libs}. For each $L \in S_{libs}$ we calculate, $DScore_L = 1 - \frac{n_L}{N}$, where n_L is the number of jobs in J that use library L. From the plot in Fig. 9.1, it can be seen 92.6% libraries *does not* appear in other 90% jobs. In fact, 68% libraries *only* appear in 1% jobs, making these libraries distinct, and thus a perfect classifier.

Further, we investigated what are the actual statistics of static and shared libraries in production clusters. We kept our analysis limited to pre-installed apps and tools (including MPI, visualization tools, etc.) as we had no access to users' home directories. On *Conte*, 2283 out of 2675 were shared libraries. On *Cab*, 1810 out of 3432 were shared libraries for pre-installed applications and for tools, 1870 out of 2821 were shared libraries, indicating abundant use of shared libraries in production clusters. Although, static libraries can be tracked using advanced tools like XALT [2] to provide better classification accuracy, the our low-cost technique supported by the reality check makes it widely applicable.

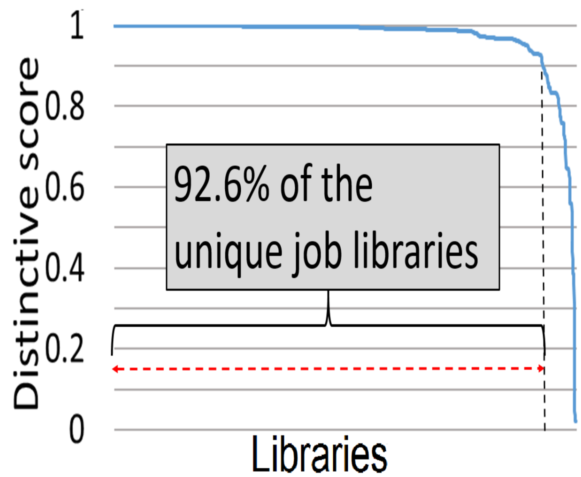

Figure 9.1. How distinctive are the libraries corresponding to unique jobs in *Conte*. Higher score is better.

Inferring failure reasons from exit codes: In Sec. 5.2 we classified the reasons for job failures by analyzing exit codes, consulting sysadmins, and various documents. We further validated some of those categories by analyzing the syslog messages. Concretely, we identified *out of memory (OOM)* and *file path or permission* related messages in the syslog and calculated how strongly those messages can be associated with jobs failing with corresponding exit code. We found, for 92% of the jobs that failed with exit codes representing memory exhaustion, was also preceded by syslog with OOM related messages. Calculating the reverse association, we found 77% of the jobs that output the related message, ultimately exited with exit codes for memory exhaustion. Similarly, exit codes denoting file path or permission problem were preceded by relevant error messages 41% of the time. A drawback of the exit code based or scheduler dependent failure analysis is that more fine-grained failure classification cannot be achieved. A thorough analysis of syslog might unravel more nuanced causes of failure and we leave this as future work.

Further, we also present a scoring mechanism (*FScore*), using which we show that few (17) libraries have high scores with respect to memory failure. At this point, we also acknowledge that due to unavailability of the ground truth in the analyzed data set, it is probable that some high scoring libraries are bug-free and the ones that get lesser score actually need attention. And this concern exists with any other scoring method as well until a reality check is performed. Therefore, the key takeaway we have from *FScore* scheme is that such a scoring mechanism could be designed and validated (or appropriately modified) by cluster support team by investigating the suspicious libraries. And by doing so pro-actively, a better quality of service is possible and consequently, a lesser job failures.

10 RELATED WORK

We classify related works into following major categories.

Modeling and benchmarks: Modeling application workload is important for efficient resource usage and scheduling. In the context of supercomputers, [17–19] presents modeling techniques for scientific applications. [20,21] discuss modeling workloads for mainstream commercial systems where the application characteristics is considerably different from scientific applications. These are orthogonal to our work as we focus on how workload analytics can help in cluster management by identifying hidden trends in resource usage and user behavior.

Performance characteristics: Large body of work, such as [22–25] study workload characteristics in order to optimize performance on different architectures. [6,26] study performance characteristics in the context of energy and other resource usage prediction. [4] presents techniques to classify tasks based on the analysis of performance metrics, and its application in capacity planning and job scheduling. [5] presents a characterization of MapReduce cluster running Hadoop workloads. Our study involves large community cluster workloads generated by a much more diverse set of applications from multiple domains. Moreover, we present library usage based application classification technique which is simple, yet powerful, especially when privacy and security is a prime concern.

Failures: Several studies, [27–29] evaluated failures in large scale systems. [7,8] analyzed failures in HPC clusters. [9] further characterizes application resilience related to system errors. The customer support centric focus of our work is unique as we show how systematic analysis of performance issues and job failures can be used to proactively help and educate users. We also attempt to understand user behavior, resource wastage or bottlenecks. The recommendations we provide based on these analysis can help in better cluster management and resource planning.

11 CONCLUSION

We demonstrate how cluster management, job performance and user experience can be improved through a user centric analysis of workloads collected from a university wide community cluster and a large supercomputing center in a government lab. We show how our analysis unraveled some hidden issues regularly faced by the users. We identify a need for special user-training on resource allocation, scheduling policies and better memory management techniques. We also propose data analysis techniques that can be used by a proactive user support system to improve service quality by pre-installing popular applications and libraries, giving attention the premium users, and helping users who have jobs with memory-related issues. Some of our techniques and results are also useful in capacity planning for future systems. Moreover, after proper anonymization, we have made the workload data available for the community through an open repository to help further research.

REFERENCES

REFERENCES

[1] Todd Gamblin, Matthew LeGendre, Michael R Collette, Gregory L Lee, Adam Moody, Bronis R de Supinski, and Scott Futral. The spack package manager: bringing order to HPC software chaos. In *Supercomputing (SC)*, 2015.

[2] Kapil Agrawal, Mark R Fahey, Robert McLay, and Doug James. User environment tracking and problem detection with XALT. In *Workshop on HPC User Support Tools (HUST)*, 2014.

[3] A open repository for cluster workload data. `https://diagrid.org/resources/247`.

[4] Asit K Mishra, Joseph L Hellerstein, Walfredo Cirne, and Chita R Das. Towards characterizing cloud backend workloads: insights from google compute clusters. *SIGMETRICS Performance Evaluation Review*, 2010.

[5] Zujie Ren, Xianghua Xu, Jian Wan, Weisong Shi, and Min Zhou. Workload characterization on a production hadoop cluster: A case study on taobao. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2012.

[6] S Huang and W Feng. Energy-efficient cluster computing via accurate workload characterization. In *International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, 2009.

[7] Bianca Schroeder and Garth A Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2010.

[8] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *Dependable Systems and Networks (DSN)*, 2014.

[9] Catello Di Martino, William Kramer, Zbigniew Kalbarczyk, and Ravishankar Iyer. Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 HPC application runs. In *Dependable Systems and Networks (DSN)*, 2015.

[10] Moab: Administrators Guide. `http://docs.adaptivecomputing.com/mwm/archive/6-0/MWMAdminGuide.pdf`.

[11] Todd Evans, William L. Barth, James C. Browne, Robert L. DeLeon, Thomas R. Furlani, Steven M. Gallo, Mathew D. Jones, and Abani K. Patra. Comprehensive resource use monitoring for HPC systems with tacc stats. In *HPC User Support Tools (HUST)*, 2014.

[12] Troy Baer and Doug Johnson. pbsacct: A workload analysis system for pbs-based HPC systems. In *Annual Conference on Extreme Science and Engineering Discovery Environment(XSEDE)*, 2014.

[13] Linux tool: List of Open Files.
`https://people.freebsd.org/~abe/`.

[14] Deniz Ersoz, Mazin S Yousif, and Chita R Das. Characterizing network traffic in a cluster-based, multi-tier data center. In *International Conference on Distributed Computing Systems (ICDCS)*, 2007.

[15] Top applications.
R: `http://www.r-project.org`,
Devel: `https://www.drupal.org/project/devel`,
Matlab: `http://www.mathworks.com/products/matlab`,
NEMO5: `https://nanohub.org/groups/nemo5distribution`,
LAMMPS: `http://lammps.sandia.gov`,
NCL: `http://www.ncl.ucar.edu`,
SRA-Toolkit: `http://www.ncbi.nlm.nih.gov/Traces/sra/`,
Gromacs: `http://www.gromacs.org`,
BWA: `http://bio-bwa.sourceforge.net`,
TopHat: `http://ccb.jhu.edu/software/tophat`.

[16] Pang-Ning Tan, Michael Steinbach, Vipin Kumar, et al. *Introduction to data mining*, volume 1. Pearson Addison Wesley Boston, 2006.

[17] Walfredo Cirne and Francine Berman. A comprehensive model of the supercomputer workload. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2001.

[18] Hui Li, David Groep, and Lex Wolters. Workload characteristics of a multi-cluster supercomputer. In *Job Scheduling Strategies for Parallel Processing*. Springer, 2005.

[19] Mustafa M Tikir, Laura Carrington, Erich Strohmaier, and Allan Snavely. A genetic algorithms approach to modeling the performance of memory-bound computations. In *Supercomputing (SC)*, 2007.

[20] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *SIGMETRICS Performance Evaluation Review*, 1998.

[21] Ludmila Cherkasova and Peter Phaal. Session-based admission control: A mechanism for peak load management of commercial web sites. *IEEE Transactions on Computers*, 2002.

[22] Laura C Carrington, Michael Laurenzano, Allan Snavely, Roy L Campbell, and Larry P Davis. How well can simple metrics represent the performance of HPC applications? In *Supercomputing (SC)*, 2005.

[23] Richard Murphy. On the effects of memory latency and bandwidth on supercomputer application performance. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2007.

[24] Lei Chai, Qi Gao, and Dhabaleswar K Panda. Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In *International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, 2007.

[25] Kevin Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant Patel, Trevor Mudge, and Steven Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. In *International Society for Computers and Their Applications (ISCA)*, 2008.

[26] Rubing Duan, Radu Prodan, and Thomas Fahringer. Performance and cost optimization for multiple large-scale grid workflow applications. In *Supercomputing (SC)*, 2007.

[27] Ramendra K Sahoo, Mark S Squillante, Anand Sivasubramaniam, and Yanyong Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *Dependable Systems and Networks (DSN)*, 2004.

[28] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production mapreduce cluster. In *International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, 2010.

[29] Derrick Kondo, Bahman Javadi, Alexandru Iosup, and Dick Epema. The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems. In *International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, 2010.