A Study of Failures in Community Clusters: The Case of Conte

Subrata Mitra[*†], Suhas Javagal[*†], Amiya K. Maji[†], Todd Gamblin[‡], Adam Moody[‡], Stephen Harrell[†], Saurabh Bagchi[†]
[*]Equal Contributors [†] Purdue University [‡]Lawrence Livermore National Laboratory

***Abstract*—Large community clusters are becoming common in universities and other organizations due to the benefits they provide to participating researchers in terms of reduced operational costs and a bigger resource pool. However, effective management, and diagnosing failures and performance issues in these clusters are challenging tasks due to the diversity of workloads run by users from various domains and experience levels. Many users who use these clusters have very less experience in computing and hence often face performance issues — leading to resource wastage. In this paper, we study these dynamics in one of the largest university-wide community clusters. We perform in-depth analysis of library and application usage patterns, job failures and performance issues. Further, we introduce a set of novel analysis techniques that can be used to identify hidden trends and diagnose job failures in compute clusters in general. We provide concrete recommendations for the cluster administrators and present case studies highlighting how such information can be used to proactively solve many user issues, ultimately leading to better quality of service.**

I. Introduction

Over the last several years, community clusters have gained immense popularity in leading research universities across the globe (examples being RCAC at Purdue, CIRC at Rochester, ITRC at University of Delaware, and HPCCC at Auburn University etc.). Community cluster movement is fundamentally driven by the advantages of resource consolidation. Instead of university research groups managing their own small clusters (typically 10-20 nodes), they contribute their funding towards buying nodes in the community cluster and, thereby, build a much larger cluster with state-of-the-art hardware. By contributing to the community cluster, research groups get dedicated access to the bought nodes and also offload the management efforts to a centralized organization. This model is not only efficient in terms of datacenter space, but also gives much greater redundancy in terms of spare hardware for each research group, allowing them to opportunistically use unused resources of the cluster. Despite these advantages, community clusters pose new challenges for cluster administrators because of the diversity of users, applications and resource usage patterns. For reliability researchers, this new ecosystem create interesting candidates for failure studies.

Our observations indicate that administration of these community clusters is distinctly difficult due to three key factors: a) *Diversity of users* not only in terms of their research focus but also in terms of their experience level and computational knowledge, b) *Diversity of workloads* in terms of resource requirements and research focus, and c) *Limited no. of support staff* who have to perform a wide variety of tasks, such as, system maintenance, application installation, troubleshooting, as well as user training. These challenges call for the needs to develop better automation and novel workload analytics techniques to improve cluster reliability and efficiency, which is one of the objectives of this paper. It is traditionally known in the research community that the reliability and performance characteristics of scientific applications not only depend on their input datasets, but also on their execution environments and runtime configurations [1]. This is especially important in a community cluster environment where each individual machine is often shared between multiple users' jobs to improve utilization, thereby, creating the potential for one job adversely impacting another. We found several instances where the inexperience of few users (a.k.a., *the human factor*) contributed to a significant fraction of failures. Finding the *what*s and *why*s of these failure modes is a non-trivial task and can pose significant challenges for failure localization and performance debugging. In this paper, we highlight these challenges for future software reliability research.

In scientific literature there have been several work on workload characterization [2], [3], [4] and failure analysis [5], [6], [7] of large clusters in general. While Martino *et al.* [6], [7] characterized application resilience related to system errors (at Blue Waters cluster), Schroeder *et al.* [5] analyzed distribution of failure rates and repair times of various clusters (at Los Alamos National Lab). To our knowledge, ours is the first attempt to understand the dynamics of a *community cluster* and its failure modes based on real data collected from a production community cluster (Conte at Purdue University, which ranks within top 100 supercomputers across the globe [8]). Due to the distinctions and challenges described above, failure modes in community clusters are significantly different from those in traditional supercomputers [7] and therefore, require rigorous analysis. We believe such analysis can help improve the quality of system administration and reliability of existing community clusters and also help in designing better clusters in future.

To systematically analyze the failures in community clusters, we lay out the objectives of this paper as follows:

1) Identify applications and libraries that are most popular among the users
2) Understand primary failure modes for jobs in community clusters
3) Develop a lightweight technique for classifying a job to fine-grained application groups
4) Develop a technique for finding potential buggy applications based on failure and application usage statistics

While the first two objectives help cluster administrators to focus their attention on critical issues and applications, the last two objectives help in building better automation for failure diagnosis. To achieve the objectives listed above, we developed a set of novel tools and techniques that can provide useful insights from various logs collected regularly in these clusters such as scheduler accounting logs, library lists from jobs, anonymized set of job-scripts, and performance monitoring data using TACC-stats. Our dataset comprises of workload traces for $489,971$ jobs from 306 users during the period of Oct 2014 and Mar 2015 from Purdue's *Conte*, one of the largest university-wide community clusters in the world. *Conte* is a

hybrid supercomputer (with Intel Xeon processors and Xeon-Phi coprocessors) with a peak performance of 977 TFlops [8]. Another interesting feature about *Conte* is its diverse user base consisting of students, faculty, and staff from 27 distinct departments from Schools of Science, Engineering, Technology, and Communications in the university. The user diversity of *Conte* makes its dataset representative of other such community clusters. A full summary of the dataset is available in Table I.

Cluster Name	Dataset Duration	No. of Jobs	No. of users	Unique application behaviors
Conte [8]	Oct 2014–Mar 2015	489,971	306	3,373

TABLE I: Summary of workload data analyzed

Concretely, our paper makes the following contributions:

1) A novel set of techniques for analyzing cluster workloads:
   a) Fine-grained classification of applications based on shared libraries used,
   b) Analyzing reasons for job failures using exit codes and syslog messages, and
   c) Statistically tracing failure root causes to libraries,
2) Detecting anomalies in memory usage and other resources;
3) Validating the utility of our techniques through discussions with users
4) We released an anonymized version of the failure data via an open repository for use by the community [9].

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

1. Some libraries are overwhelmingly popular across different applications and users—top 10 of 3080 unique application libraries are used by 40% of the users. These can be hand-optimized, replicated and placed in local storage, and pre-installed on these systems.

2. Among the failure categories, jobs exceeding walltime and memory issues are dominant modes. Sharing of nodes with other users and incorrect resource estimation are primary causes of out of memory error.

3. A few users contribute to disproportionately high number of major page faults. We found that the top 10 users of the system contribute to 50% of the jobs that face high page faults.

4. High page faults lead to wastage of compute time on the clusters. We also found high-page faults to be strongly correlated with jobs exceeding walltime. This poses interesting research problems for improving cluster efficiency.

The rest of the paper is organized as follows. Sec. II describes our data sources and analysis methodology. In Sec. III, in-depth analysis of application usage is carried out. In Sec. IV we diagnose job failures. Sec. V presents few user case studies based on the findings from our analysis. Sec. VI highlights prior works and Sec. VII concludes.

II. An Overview of Data Sources and Analyses

A. Cluster Hardware and Software

Purdue's *Conte* is one of the largest university-wide community clusters. It has 580 nodes each with two 8-core Intel Xeon E5-2670 processors, two Xeon Phi accelerator cards and 64GB of memory. All nodes run RHEL-6.6 OS. The job scheduler is TORQUE 4, an open source implementation of Portable Batch System (PBS). At submission time, each job requests for a certain time duration of execution (aka walltime), number of nodes and, optionally, amount of memory needed. Job scheduling uses a community cluster allocation method in which, some research groups purchase nodes and get *semi-dedicated* access to their purchased nodes 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. By default, only a single job is scheduled on an entire node giving the job dedicated access to all the resources of the node. However, node sharing can be enabled using a PBS directives in job submission scripts. We found a significant number of jobs using a shared node environment for various reasons as highlighted in Sec. IV-E.

```
10/01/2014 00:05:55;E;1660509.machineIP.uni.edu;user=U
     group=G jobname=test1 queue=Q ctime=1412131890 ...
owner=user1@machineIP exec_host=node19/core0/core1
Resource_List.naccesspolicy=shared Resource_List.ncpus=1
Resource_List.neednodes=1:ppn=2 ...
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
```

Fig. 1: Example of an accounting log for *Conte*. Only the relevant fields are shown.

B. Data Sources and Collection Methodology

The dataset collected from *Conte* consists of five major components —

Accounting logs: The TORQUE accounting logs provide 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 (ppn), walltime limit and memory limit. It also contains the exit status of a job (denoting, whether job ended successfully, crashed, or exited due to configuration error or time limit etc.). A snapshot of the raw accounting logs is provided in Fig. 1.

TACC Stats: TACC stats [10] data provides more fine-grained resource usage profile on all the nodes used by the job. For each node, TACC Stats collects periodic snapshots (interval of 10 minutes) of various system metrics - local disk usage, Lustre filesystem usage, Infiniband and IP network traffic, process and memory statistics, etc.. In this paper, we mostly focus on memory related statistics.

Syslog: The Syslog data comprises of kernel and system messages of all nodes and is collected at a single node. The log messages relevant to our analysis are those containing memory errors, OOM killer messages, filesystem status, etc.

Library lists: The system periodically takes snapshots (every 30 minutes) of the shared libraries accessed by the jobs in each computing node using `lsof` tool [11]. Consequently, many such snapshots will be created for a long running job. A job using many nodes will have library list files generated for each node. We aggregate this information for a job from all nodes.

Job script: A user writes shell scripts to lay out the tasks a batch job would perform. Depending on the expertise of the user, the complexity of a job script varies from a simple task specified through a single command line to a pipeline of tasks with complex setup for each.

C. An Overview of Data Analyses

Fig. 2 shows a high-level overview of our data collection and data analysis methodology. After collecting raw data from various logs, the first stage in our analysis pipeline is to filter noisy data that can skew the results. It has been repeatedly observed [2], clusters usually run many extremely small jobs

Observations	Possible implications	Recommendations	Reference
O1: Some *non-preinstalled* libraries and applications are highly popular (238 or top 500 libraries).	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. III
O2: Jobs using a set of 17 libraries almost always lead to memory exhaustion failures.	These libraries might have internal memory bugs.	**R2**: We provide a technique to localize the source of such a failure.	Sec. IV, Fig. 5
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. IV-F, Sec. V
O4: 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.	**R4**: Train users about software design and use of more number of nodes to avoid memory thrashing.	Sec. IV-D
O5: 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.	**R5**: Training should cover the difference of behavior between shared and non-shared environment.	Fig. 10, Fig. 11
O6: Jobs that used $> 75\%$ of its walltime were $5X$ more likely to suffer from memory thrashing than other jobs.	Different failure types are inter-dependent, e.g., *exceeding walltime* may be a symptom for *out of memory* error.	**R6**: Thorough analysis of resource usage along with failure symptoms, as presented in this paper, can help diagnosis tools achieve better accuracy.	Sec. sec:runtime, Fig. 13

TABLE II: Highlight of the key observations, possible implications and our recommendations.

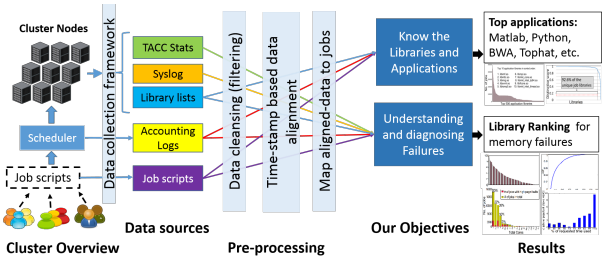

Fig. 2: Cluster workload data analyses workflow. We focus on understanding failure modes and predicting buggy libraries.

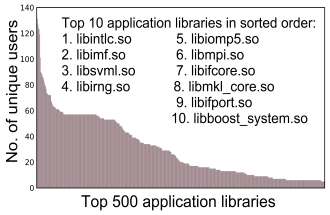

Fig. 3: Histogram of top 500 libraries used by unique users in *Conte*. Top 10 libraries are used by 40% of the users.

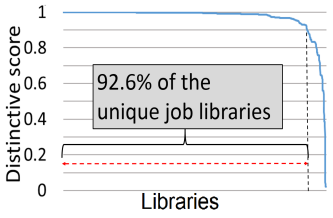

Fig. 4: Distinctive score of the libraries for unique jobs. Higher score is better.

which were either terminated by the user or exit due to configuration errors. To reduce such noise, we filtered out the jobs that run for less than 30 seconds. In the next stage, we align these datasets based on either their timestamps or their job ids (where available). More precisely, we align accounting logs, TACC Stats and Syslogs messages using the timestamps and tie-up datasets from library lists and job submission scripts using their job ids. After data cleaning and alignment (pre-processing), we apply our analysis tools to fulfill the objectives specified in Section I. At a high level, our analyses involve finding various distributions, such as, library and application distributions, failure distributions, resource usage distributions etc. and then using these distributions to detect potential buggy libraries. We also propose a novel clustering approach that can map a job to fine-grained application groups based on library usage data.

Open data repository: A thoroughly anonymized version of the workload data from *Conte* is made publicly available to the research community for further analysis through an open repository. The first release was made in November 2015 [9].

III. Understanding the Applications and the Libraries

To understand the failure modes in a community cluster, it is important to know about the applications running on it. Learning about the popular applications and libraries can also guide sys-admins to prioritize the software installation and maintenance procedures. For example, applications that are heavily used, must be carefully tested and tuned (possibly with multiple regression tests). Thanks to the user diversity, *Conte* runs a variety of applications. Consequently, these applications use various shared libraries (.so) during their execution. In this section, we focus our analysis to answer the following questions: (1) Which libraries are most heavily used? (2) Which applications are used the most? (3) Is it possible to characterize user jobs based on a fine-grained classification of its runtime environment?

A. Popular libraries:

Jobs or application use various libraries, some of which come with the OS distribution, some are pre-installed in the environment, and rest are downloaded and installed by individual users. We identify unique libraries from all the jobs and discard the default OS libraries (in */usr/lib64* and /lib64) as these would falsely skew our results. As a result we got 3,080 unique application libraries from which we created sorted histograms by counting how many times each library was used by a user, as shown in Fig. 3 with top-10 most frequently used libraries highlighted. In Fig. 3, out of 3,080 libraries, each of the top 10 libraries are used by approximately 40% of the users. We also found, if sorted based on number of times used by the jobs, the top-most library is used 4X times more than the 50^{th} one. We recommend that extracting such 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 but also minimize the risk of using a buggy or unoptimized versions which could lead to performance problems.

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*. After consulting with the cluster administrators, we learned that the pre-installed applications come in two flavors—those installed by the central IT staff and those managed by domain experts (e.g. bioinformatics, nanotechnology etc.). We loaded all these modules and listed the unique shared libraries from their LD_LIBRARY_PATH. Comparing this list with our frequently used libraries, we found that 46 of top 50, and 92 of top 100 libraries are pre-installed. Interestingly, only 262 of top 500 libraries were pre-installed. This is probably because the currently pre-installed libraries were chosen based on intuitions. Using a quantitative analysis technique such as ours can help cluster administrators make more informed library installation decisions.

B. Popular applications:

We attempted to identify what are the applications that are widely used by analyzing users' job submission scripts. In

R (Statistics)	Devel (Compilers and MPI runtime)
Matlab (Numerical Computing)	NEMO5 (Nano-Electronics Modeling)
LAMMPS (Molecular Dynamics Simulation)	NCL (Data analytics/Visualization)
SRA-Toolkit (Bioinformatics)	Gromacs (Molecular Dynamics Simulation)
BWA (Bioinformatics)	TopHat (Bioinformatics)

TABLE III: Top 10 applications used in *Conte*. The popular applications have an even mix of bioinformatics and engineering applications.

this analysis, we were able to capture only the pre-installed applications that users can use via the `module load` command (provided by the cluster environment) in their job-scripts. In Table III, we show the top 10 most heavily used out of 117 pre-installed applications in *Conte*. Users themselves can also install desired applications in their personal space. It is not possible to know these applications by only analyzing the job-scripts as the name of the executable visible in the job-script can be misleading or it can be just a wrapper on the actual application binary. Therefore, to find the actual user installed applications an access to each individual users' private space is required and is thus out of scope for this paper due to security and privacy concerns. However, through our analysis, we discovered that many of the pre-installed applications were rarely used. These applications can be replaced by other applications identified either through user survey or through the classification technique we suggest in Sec. III-C. We communicated the list of rarely-used applications to the cluster administrators which helped them in prioritizing which (outdated) applications to phase-out from the system.

C. A fine-grained application classification technique

While investigating job failures or performance issues in clusters, it is often beneficial to know which application(s) are being run as part of that job. While application name(s) can be found from the job-script data in our dataset, this information is often incomplete or insufficient. For example, an application like `MATLAB` offers many different toolboxes, such as, Statistics Toolbox, Image Processing Toolbox, Fuzzy Logic Toolbox, etc. These toolboxes not only have different resource usage patterns but also show different failure manifestations. It is not possible to distinguish these toolboxes just by looking at the `module load` commands in the job-scripts. Moreover, in clusters regulated by government agencies (national labs), privacy sensitive data such as job-scripts may not be available at all. Another use-case where jobs do not have application info is interactive batch jobs. In this type of jobs, users interactively type commands in the console instead of specifying a job-script. Therefore unless every command is logged on the cluster (which may appear privacy-invasive), application info for interactive jobs can not be inferred directly. We address these challenges by proposing a fine-grained application classification technique based on library data collected per job.

There have been prior works, which try to classify applications based on their resource usage and communication patterns [12], [2]. 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.

Classification technique - Procedure: We extract only the library names from our library-list dataset by stripping the path and version information. We sort these names (to normalize in case of address space randomization) and calculate a hash value. We consider jobs with the same hash value for the libraries are essentially running the same application (we refer to them as an *app group*).

Classification technique - Evaluation: To evaluate the accuracy of this technique 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 that the presence of only a few libraries are enough to distinguish between two otherwise similar-looking library list from two different applications.

Do unique jobs use unique libraries? To investigate the general applicability of our job classification technique, we introduce a metric: *DScore (distinctive score)* of a library. To calculate *DScore*, we identify all the *unique* job-scripts by hashing the content and treating 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. 4, it can be seen 92.6% libraries *do not* appear in other 90% jobs. In fact, 68% libraries *only* appear in 1% jobs, making these libraries distinct, and thus a perfect classifier for those jobs. From the *DScore* analysis, we conclude that the library list of a job can be used to classify its application type with high degree of accuracy.

Reality check: The *lsof* that we used to collect library data has the limitation that it only collects shared library usage statistics. We investigated in reality, what fraction of the libraries are used as shared libraries in *Conte*. Investigation was limited to pre-installed apps and tools (including MPI, visualization tools, etc.) as we had no access to users' home directories. On *Conte*, 2283 (of 2675) were shared libraries indicating abundant use of shared libraries in production clusters. Thus our simple technique supported by the reality check makes it widely applicable. Our technique would benefit if static libraries are also tracked using advanced tools like XALT [13].

IV. Understanding job failures

Understanding job failures is essential in any cluster environment to improve operational efficiency. Failed jobs translate to resource wastage and user dissatisfaction. Martino *et al.* [7] show that job failures contribute to monetary loss of as much as $421,878 in Blue Waters cluster between March 2013 to July 2014. In this section, we focus on the primary failure modes seen from the job accounting logs in *Conte*. The failure modes are determined by the exit codes recorded by the scheduler (Torque) at the time of job exit. Since we had no information about the logical correctness of the computation results or the behavior excepted by the user, we use exit codes to understand if the job faced any visible issues (or non-silent failures). We hypothesize that by looking at types of failures and causes of failures we can proactively identify actionable items to improve cluster reliability, such as, finding *applications* that are *suspected* of failures, finding *users* that can benefit from additional guidance, or finding generic *best practices* that can improve cluster utilization. In the subsequent sections, we explain how we achieve these goals.

A. Failure modes

When a job exits, the accounting log records the code with which it finishes execution. A non-zero exit status indicates an

Reason	No. of failed jobs	% of failed jobs
Time expired (timeout)	16079	20.3
Memory exhaustion	11962	15.2
Segmentation Fault	7292	9.3
Quit/keyboard interrupt	4202	5.3
File system/path problem	2901	3.7
Self abort/assert failure	474	0.6

TABLE IV: Few major types of job failures showing percentage of *failed jobs*

error with the job and is the subject of our analysis.

How to interpret job exit codes? From the TORQUE manual, we learned 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 ≥ 128 or ≥ 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 cause of an unsuccessful job termination. However, in TORQUE, exit code 1 indicates a generic error and cannot be classified to any particular category. We found that in *Conte* 16.2% of the jobs had failed (exit code not equal to 0). This is relatively higher than that reported by Martino *et al.* [7]. We speculate this may be due to lower resource usage or more experienced users or better management in [7].

What exit codes do users usually provide? We first identified 84,163 *unique* job scripts submitted to *Conte* by hashing file contents. 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 exit codes in the range [2-99] to specify error conditions. Only 5 scripts explicitly specified negative exit codes [-1,-2,-3,-20,-30] and 44 scripts explicitly used exit code 0 to mark successful completion.

Classification of failures with the help of exit codes: Using job exit codes, job's runtime and resource usage information, we classify failed jobs based on the probable reasons for failure in Table IV. We found in *Conte*, about a fifth of the *failed jobs* were killed due to exceeding the requested walltime. The next most significant factor was memory exhaustion (15.2%) followed by segmentation fault (9.3%). A small fraction of jobs were aborted by users with keyboard interrupts (5.3%). Failure reasons for 6.2% jobs could not be classified as they exited with generic code 1. 18% of the failed jobs had user provided custom exit codes between [2-99], hence cause of failure cannot be automatically identified without understanding users' intentions. There were other smaller percentage of failures of various types such as floating point exceptions, invalid memory references etc.

Discussion: For few applications, exceeding walltime may not always imply a job failure as they cannot complete within the maximum permitted walltime and are, therefore, *designed* to run up to the maximum allowed time limit. These applications often take periodic checkpoint so that it can restart from the saved state during next job. However, we found in *Conte* number of such jobs are too insignificant to make any statistical impact.

B. Correlating job failures with system logs

We further validated the classification of some of the failure categories by analyzing the syslog. 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 codes. We found, 92% jobs that failed with exit codes representing memory exhaustion, were also preceded by OOM related messages in syslog. Calculating the reverse association, we found 77% 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.

Discussion: 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.

C. Identify the suspect libraries

Among the failure categories, memory issues and segmentation faults are often related to bugs in internal libraries. Therefore, it would be beneficial to find an ordered list of *suspect libraries* to draw the attention of developers or administrators. The technique should be applicable to any type of library-dependent failures and simple enough to intuitively explain the *choice of suspect libraries*. We propose a heuristic scoring mechanism that up-weights suspicious libraries. Intuitively, such a score for a library L should consider the following factors: (1) the score is computed for a specific type of error (e). So, different errors would have different list of suspicious libraries. (2) what *fraction* of jobs associated with library L, fail due to error (e). (3) how *many* such failures we have seen.

We introduce our scoring metric *FScore* as follows. Let $\boldsymbol{Jobs}_L$ denote the set of jobs that use library L and have either exit code 0 (successful) or exit codes e (for example, e can be the error code for memory exhausion). And then let $\boldsymbol{Jobs}_L^e$ denote the set of jobs that use library L and have the exit code e. Then *Fscore* w.r.t. error e is defined as,

$$FScore^e = relative\_frequency\_of\_failures \times total\_failures$$
$$= \frac{|\boldsymbol{Jobs}_L^e|}{|\boldsymbol{Jobs}_L|} \times |\boldsymbol{Jobs}_L^e| = \frac{|\boldsymbol{Jobs}_L^e|^2}{|\boldsymbol{Jobs}_L|}$$

Suspicious libraries for detailed investigation can be chosen by selecting the top K, ordered (high-to-low) by the *FScore*. We will enunciate this technique further using the following example. Consider a set of 100 jobs. Let us say that these 100 jobs use libblas.so and only 50 of them also use libboost.so (assume libboost.so is buggy and causes memory exhaustion). If all the 50 jobs using libblas.so and libboost.so fails, then *FScore* for libblas and libboost will be 25 ($50\% \times 50$) and 50 ($100\% \times 50$) respectively. By our scoring we will have libboost ranked higher and consequently as the suspect. Fig. 5 shows the *FScore* distribution for the jobs that failed due to memory exhaustion in *Conte*. From the figure, we see that only a small set of 17 libraries can be isolated with really high FScores. 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 as Valgrind. Thus, using this scoring technique, the search for a culprit can be narrowed

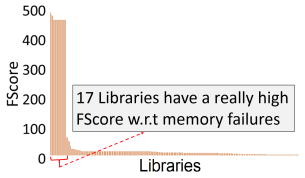

Fig. 5: Distribution of *FScore* values w.r.t failures due to memory problems.

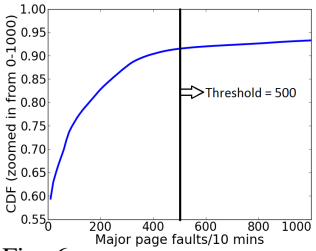

Fig. 6: CDF of peak major page fault of all jobs, threshold set at 90^{th} percent value.

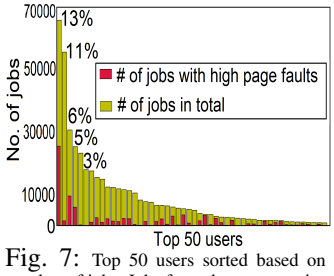

Fig. 7: Top 50 users sorted based on number of jobs. Jobs from the top user significantly suffered from major page faults.

down to only a few libraries.

This scoring mechanism, however, has two disadvantages: (1) It may miss out some of the rarely used yet buggy libraries because the multiplicand in our *FScore* expression up-weights the library if it is associated with *many* failed jobs. This is an artifact of our policy decision as we wanted to focus only on the *frequently used* libraries which might be buggy. Different scoring functions can be used to explore other types of ranking. (2) We assume that jobs with an exit code of 0 exit with success. This is not true if the last statement in a job script is successful but there are one or more errors prior to that (in this case the scheduler believes that the job exited successfully). Analyzing such errors would require detailed logs from jobs (not just accounting logs) and is therefore beyond the scope of this paper. On the contrary, a non-zero exit code (memory error or segmentation fault) is definitive indicator of failure. Our analysis is, therefore, *optimistic* in that it assumes a library is error-free in the absence of hard evidence.

D. The issue of memory pressure

In Table IV, we found that memory errors contribute as the second most frequent failure category. In general, memory usage related issues are frequently responsible 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[1]. 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. Our analysis in Table IV consider these kinds of job failures. In this section, we characterize the impact of memory thrashing. 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 during initialization and start-up, we exclude first 10 minutes of its run-time from our analysis to discard start-up noise.

We define ***peak major page fault*** of a job as the maximum

[1]If a node has no local disk, then the application would crash with out-of-memory exception

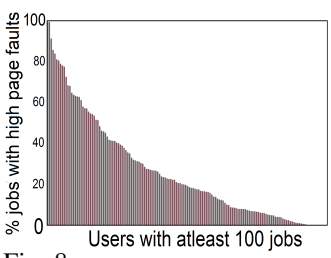

Fig. 8: Percentage of user's jobs crossing peak major page fault threshold. Users with high page fault need assistance from support staff.

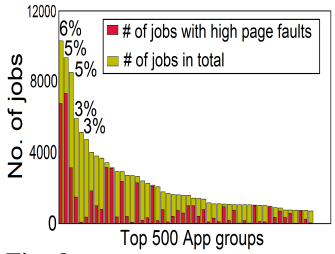

Fig. 9: Sorted histogram of app groups w.r.t number of jobs crossing peak major page fault threshold. Top-2 app groups suffer frequently from page faults.

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 wanted to be conservative in identifying such cases as we did not have the ground truth. First, we identify a threshold for peak major page fault by calculating the CDF across all the jobs (Fig. 6). It can be seen that 90% of the jobs experience less than 500 major page faults within any 10-minute time window, we use this as our threshold for all subsequent analysis. Fig 7 show what percentage of jobs submitted by top-50 active users suffered from severe page faults (*i.e.,* peak page fault was more than the computed threshold). 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. 8 we show the histogram of jobs for unique users sorted based on the percentage of jobs that suffered severe 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, as discussed in Sec V.
We also analyzed what percentage of the jobs in the most popular app groups (sorted based on number of jobs) suffer from frequent major page faults. This is presented in Fig. 9. To our surprise, we found that top two app-groups suffer from major page fault quite severely.
Lessons learned: 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, such as using *weak-scaling* [14] methods to distribute memory usage over a large number of nodes to avoid memory exhaustion or use of *generators* and *yield* like features in Python and other advanced programming languages [15] to reduce the memory footprint of the application. The support staff might also pre-install a properly optimized and memory leak free version of these applications, if not already available.

E. Factors impacting memory pressure

In the previous section, we showed *how* to detect the *users and applications* that suffer from major page faults using simple histograms. In this section, we try to analyze the *whys* of memory issues, more specifically we focus on the *human-centric* reasons for severe thrashing. The factors that were found to cause memory issues are: (1) Sharing nodes with other users and (2) Incorrect geometry of multi-process (MPI or OpenMP) jobs. We present a systematic analysis of these

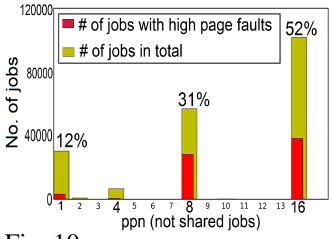

Fig. 10: Non-shared env: effect of ppn on major page faults. As ppn increases more jobs suffer from severe page faults.

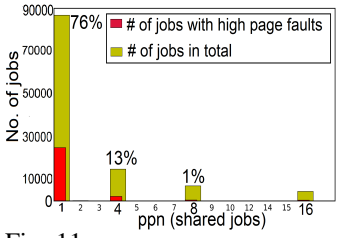

Fig. 11: Shared env: effect of ppn on major page faults. As ppn increases fewer jobs suffer from severe page faults.

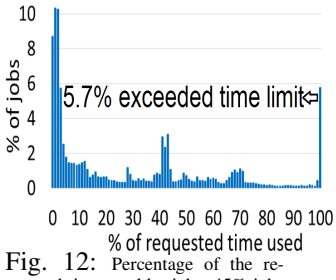

Fig. 12: Percentage of the requested time used by jobs. 45% jobs used less than 10% of the requested time and 5.7% exceeded time limit.

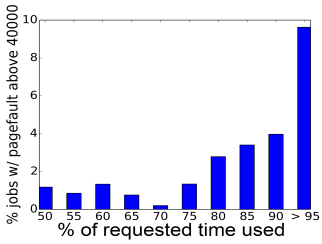

Fig. 13: Major page faults is correlated with percentage of requested time used by a job (beyond 75% of requested time).

two factors and show how often jobs suffer from page faults due to these. Based on this analysis, we come up with some generic suggestions for users to alleviate such issues.

Sharing of nodes: By default, in *Conte* jobs do not share a node. However, users can enable sharing by setting `naccesspolicy=shared` in the 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. The key reasons were: 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.

Effect of process placement: As discussed earlier, user can set the no. of cores per node through the parameter `ppn` in the PBS script. We analyzed how the choice of `ppn` affects the memory usage issues for both non-shared and shared environments (in Fig. 10 and 11). 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 more processes on a node impose greater memory pressure. On the other hand, in a shared environment, as `ppn` increases, percentage of jobs with high page faults decreases because the level of interference from other co-located applications decreases. There are some other production clusters where in shared mode, the memory is strictly allocated in proportion to the requested `ppn`. Future work can explore whether workloads on those clusters would also exhibit similar behavior.

Lessons learned: The observations in Fig. 10 and 11 introduces additional challenges for performance debugging and auto-tuning (applications that try to predict good configurations at runtime) in a shared cluster, since memory pressure arising from other applications must be considered. To reduce memory pressure we also suggested users to use more restrictive node-access policies, such as `naccesspolicy=singleuser` (co-locates multiple jobs from a single user) or `naccesspolicy=singlegroup` (co-locates multiple jobs from users of a single group), for shared jobs. It is also essential to profile memory requirement per-core to find good process-placement for multi-core jobs.

F. The issue of runtime estimation

Overestimating job runtime: In Table IV, we found that the topmost failure category in *Conte* is jobs killed due to timeout. However, to our surprise, we found almost 45% of jobs used less than 10% of the requested times in *Conte* (Fig. 12), while around 5.7% of the jobs exceeded the requested time. This indicate that incorrect runtime estimation by the user can lead to significant wastage of computation resources. In terms of scheduling, we also found longer jobs took longer time to schedule (we omit the plots due to page budget). Therefore, overestimating job duration can impact the user in terms of total time (wait time in queue + execution time).

Jobs exceeding requested time: On the other end of the spectrum, underestimating runtime leads to jobs being killed prematurely. We wanted to verify if job timeouts are only due to user error or can be caused by other factors. We hypothesize that the jobs that run out of their requested walltime may have experienced massive memory thrashing, therefore, resulting in slowdown. In Fig. 13, we plot percentage of jobs experiencing severe page faults (peak page fault was more than an empirically derived threshold of $40,000$ page faults in any 10 minute interval) against the percentage of requested time actually used. We found that the number of jobs facing such massive major page faults is almost *5X* higher for the jobs that used up $> 95\%$ of requested time, than the jobs that 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.

Lessons learned: This analysis highlights how various types of failures are dependent on one another. While jobs exceeding walltime is the manifestation, this can be caused by memory issues. Therefore, automated tools for failure diagnosis must have complete knowledge of various resource utilization, as well as, knowledge about dependencies across failure types.

V. Case Studies

We now present a summary of few case studies where we identified the problem through our analysis, interviewed the concerned users and suggested remedies. In all the cases, we *proactively* contacted the users and they acknowledged that they are facing issues related to their jobs.

Too many processes per node: Three users faced similar problems caused because of having too many processes per node, causing physical memory exhaustion and high page faults. For example, one user observed out-of-memory errors when migrating from a third party app to in-house code, mimicking the same functionality but at a larger scale. He did increase the no. of processes but could not increase no. of nodes as he was using his group's dedicated queues. The issue was alleviated when we suggested to use shared queue and decrease the `ppn` but increase the number of nodes.

File append: A user started experiencing extreme slowdown (run-time of $\sim$10hrs instead of $\sim$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 a file and then performing file seek to arbitrary locations inside 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.

VI. Related work

We classify related works into following major categories.

Failure analysis: Several studies, [16], [17], [18] evaluated failures in large scale systems. [5], [6] analyzed failures in HPC clusters. [7] further characterizes application resilience related to system errors. Our analysis is orthogonal to these previous research in two respects: first, we analyze failures in a large community cluster where workloads are much more diverse and users have wide range of experiences. Second, in addition to correlating failures with buggy libraries, we focus on failures originating from human error and lack of experience. We also provide concrete recommendations that would alleviate such failures.

Debugging performance problems: Recent research produced statistical tools [19], [20], [21], [22] that, in the presence of sufficient historical data, can diagnose the root cause of a bug. There are some dedicated tools to diagnose performance problems in MPI applications [23], [24], [25] either by tracking anomalous data movements, difference in progress through control-flow or detecting deadlocks. These tools require many customized and fine grained information from the applications such as control-flow information, detailed execution logs, timing information etc. In contrast, our approach for identifying suspicious libraries is simple and relies on the data that are regularly collected by sys-admins. Nevertheless, the above mentioned tools and techniques are complementary and would be required for in-depth analysis of the actual root-cause of the failure. Our analysis can guide the design of these tools by highlighting the impact of node sharing and process placement.

VII. Conclusion

In this paper, we presented a study of job failures and performance issues in a large university-wide community cluster (Conte). To the best of our knowledge, this is the first failure analysis of a production community cluster based on real failure data. We found that, due to the wide diversity of research domains and user experience levels, community clusters have an interesting mix of user errors and application errors. More specifically, we found that the largest failure categories were those related to inaccurate estimation of job runtime and memory issues. Even though these two failure types appear independent, we found strong correlation between memory issues and jobs exceeding walltime. We also presented two lightweight techniques for improving cluster reliability: (1) a fine-grained classification technique for mapping jobs to various application groups, and (2) statistically predicting potential buggy libraries based on job failure data. We also presented some of the affected users with suggestions for alleviating these issues. Our future work would involve a thorough analysis of the collected syslogs and resource usage data (such as network usage and Lustre IO metrics) to find more fine-grained reasons for failures and performance issues.

References

[1] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, *A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 115–131.

[2] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, "Towards characterizing cloud backend workloads: insights from google compute clusters," *SIGMETRICS Performance Evaluation Review*, 2010.

[3] Z. Ren, X. Xu, J. Wan, W. Shi, and M. Zhou, "Workload characterization on a production hadoop cluster: A case study on taobao," in *IISWC*, 2012.

[4] S. Huang and W. Feng, "Energy-efficient cluster computing via accurate workload characterization," in *CCGrid*, 2009.

[5] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE TDSC*, 2010.

[6] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *DSN*, 2014.

[7] C. D. Martino, W. Kramer, Z. Kalbarczyk, and R. Iyer, "Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 hpc application runs," in *DSN*, 2015.

[8] "Conte among the top 500 supercomputers in the world," https://www.top500.org/system/178084.

[9] "A open repository for cluster workload data," https://www.rcac.purdue.edu/fresco/readme.html.

[10] T. Evans, W. L. Barth, J. C. Browne, R. L. DeLeon, T. R. Furlani, S. M. Gallo, M. D. Jones, and A. K. Patra, "Comprehensive resource use monitoring for hpc systems with tacc stats," in *HPC User Support Tools (HUST)*, 2014.

[11] "Lsof: List of Open Files," https://people.freebsd.org/~abe/.

[12] D. Ersoz, M. S. Yousif, and C. R. Das, "Characterizing network traffic in a cluster-based, multi-tier data center," in *ICDCS*, 2007.

[13] K. Agrawal, M. R. Fahey, R. McLay, and D. James, "User environment tracking and problem detection with xalt," in *Workshop on HPC User Support Tools (HUST)*, 2014.

[14] "Weak scaling in HPC," https://computing.llnl.gov/tutorials/parallel_comp/.

[15] "Generators: Reducing memory consumption," https://wiki.python.org/moin/Generators.

[16] R. K. Sahoo, M. S. Squillante, A. Sivasubramaniam, and Y. Zhang, "Failure data analysis of a large-scale heterogeneous server environment," in *DSN*, 2004.

[17] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production mapreduce cluster," in *CCGrid*, 2010.

[18] D. Kondo, B. Javadi, A. Iosup, and D. Epema, "The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems," in *CCGrid*, 2010.

[19] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *OSDI*, 2012.

[20] T. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," in *ICSE*, 2009.

[21] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *PLDI*, 2012.

[22] I. Laguna, S. Mitra, F. A. Arshad, N. Theera-Ampornpunt, Z. Zhu, S. Bagchi, S. P. Midkiff, M. Kistler, and A. Gheith, "Automatic problem localization via multi-dimensional metric profiling," in *SRDS*, 2013.

[23] Q. Gao, F. Qin, and D. K. Panda, "Dmtracker: Finding bugs in large-scale parallel programs by detecting anomaly in data movements," in *SC*, 2007.

[24] S. Mitra, I. Laguna, D. H. Ahn, S. Bagchi, M. Schulz, and T. Gamblin, "Accurate application progress analysis for large-scale parallel debugging," in *PLDI*, 2014.

[25] W. Haque, "Concurrent deadlock detection in parallel programs," *International Journal of Computers and Applications*, pp. 19–25, 2006.