Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 09-018

Extracting the Textual and Temporal Structure of Supercomputing Logs

Sourabh Jain, Inderpreet Singh, Abhishek Chandra, Zhi-li Zhang, and Greg Bronevetsky

June 01, 2009

Extracting the Textual and Temporal Structure of Supercomputing Logs

Sourabh Jain*, Inderpreet Singh*, Abhishek Chandra*, Zhi-Li Zhang* and Greg Bronevetsky†
*Department of Computer Science, University of Minnesota-Twin Cities, Minneapolis, MN 55455
Email: {sourj,isingh,chandra,zhzhang}@cs.umn.edu
†Lawrence Livermore National Laboratory, Livermore, CA 94551
Email: greg@bronevetsky.com

***Abstract*—Supercomputers are prone to frequent faults that adversely affect their performance, reliability and functionality. System logs collected on these systems are a valuable resource of information about their operational status and health. However, their massive size, complexity, and lack of standard format makes it difficult to automatically extract information that can be used to improve system management. In this work we propose a novel method to succinctly represent the contents of supercomputing logs, by using textual clustering to automatically find the syntactic structures of log messages. This information is used to automatically classify messages into semantic groups via an online clustering algorithm. Further, we describe a methodology for using the temporal proximity between groups of log messages to identify correlated events in the system. We apply our proposed methods to two large, publicly available supercomputing logs and show that our technique features nearly perfect accuracy for online log-classification and extracts meaningful structural and temporal message patterns that can be used to improve the accuracy of other log analysis techniques.**

I. Introduction

Supercomputers are complex machines built from large numbers of components. Although these components may be individually reliable, when aggregated at the scale of hundreds to tens of thousands of nodes, the probability of individual component failures and destructive multi-component interactions becomes significant. In particular, large scale machines such as the ASCI Q and Red Storm have been unusable by applications 17% and 26% of the time, respectively, due to system problems [6]. Further, typical mean times between failures are on the order of hours or days [6], [16], which results in 10-20 application restarts per day on machines such as Red Storm, Purple and BlueGene/L [5]. These faults can occur in many different system components, including the network, the scheduler, the file system, compute nodes and applications. Moreover, due to the complex interactions between different components, many of these faults are related to each other, with faults in one location/component affecting other parts of the systems in unanticipated ways. Such complex interactions make the identification, prediction and localization of faults extremely difficult.

Systems routinely collect information about the state of their software and hardware, including fault notifications, informational status updates as well as behavioral parameters such as temperature. This information is collected in various forms such as system, application and console logs [9], as well as RAS Databases [12]. Therefore, these logs can provide valuable information for identifying and locating system faults. However, the large size of these logs makes it nearly impossible for operators to interpret them manually, which severely limits their usefulness for managing supercomputing systems. For example, the log from the BlueGene/L system at Lawrence Livermore National Laboratory (LLNL) (Table I) [2] covers 215 days of operation time, contains over 4 million messages generated at a rate of 15 messages/minute, of which over 348,460 are alerts. Furthermore, these log-messages lack any standard format or semantics. Instead, log entries are generally plain-text messages in natural language written by a wide variety of developers from different organizations. This complicates the task of writing automated scripts to extract the useful information from these log messages.

These issues of system log size and complexity make it critically important to develop more robust and effective tools to analyze system logs. Our work is motivated by the following research questions:

• *How can we summarize and standardize the information contained in the logs?* One of the main reasons for the large size of system logs is the repetition and duplication of log messages. In practice, the number of different log message structures is quite small and most log messages are instantiations of these major structures. In this paper, we use a textual clustering technique to extract these structures and remove redundant information, thus reducing log size and complexity.

• *Can we automatically classify log-messages into semantic categories?* A key problem with system logs is the lack of any embedded semantic information, which has to be determined manually by system administrators with considerable effort. In this paper, we use an online textual clustering algorithm to semi-automatically classify log-messages into administrator-defined groups. When applied to two publicly available supercomputing logs our technique classified log-messages with more than 99% accuracy with minimal involvement from the system administrator.Furthermore, our method has a very small computation overhead and memory footprint, which is critical given the massive rate at which supercomputers generate logs.

• *How to identify temporally correlated events?* Supercomputers exhibit various kinds of events that may be correlated due

to interactions between the system components that generated them. As such, these correlations represent a valuable source of information about the system's operation. We present a technique to automatically identify these interactions by clustering groups of log messages using their temporal proximity.

Section II describes the characteristics of supercomputing logs that we have focused on in this work. Section III presents our textual log structure extraction method and Section IV covers the temporal correlation analysis. We then conclude with a discussion of related work in Section V and a summary of our contributions Section VI.

II. Characteristics of Supercomputing Logs

In this section we investigate the characteristics of supercomputing logs by analyzing two such publicly available logs. We start by describing the data-sets we used and then present the key insights gained through our analysis.

A. Data-set Description

The logs used in this study come from the following supercomputers: i) BlueGene/L supercomputer at Lawrence Livermore National Labs (LLNL) and ii) Spirit supercomputer at Sandia National Laboratories (SNL) (both available from the Supercomputer Event Log repository [2]). These logs are available for the durations of 215 days and 558 days respectively. The BG/L (*bgl*) logs were collected using the Machine Management Control System (MMCS) and have a time granularity of up to a microsecond. Collection of logs on Spirit was done using syslog-ng with time granularity of a second. Table I shows the summary description for these logs, which are described in more detail by Oliner and Stearley [13].

Each line in these log collections contains a time-stamp, a message category, the source system component as well as a textual *'message'* that describes some system event. Multi-line messages, where a single semantic message is broken up among multiple lines, are treated as multiple messages. Table II shows sample lines from the *bgl* log. Both the logs used in this study were "tagged" by the operators, by running "regular-expression"-based scripts to associate log-messages with operator-defined categories of alerts. The *bgl* log contains 41 such categories while the *spirit* log has only 8. These regular expressions were written after in-depth manual analysis, and hence closely reflect the ground truths.

The textual message in each line of the logs is composed of multiple tokens, which are space-separated strings. We divide these tokens into three categories: i) Words: tokens with only alphabetic characters. ii) Numbers: tokens that represent decimal or hexadecimal numbers. iii) Other symbols: tokens that fall in neither category, such as IP addresses, component names, etc.

B. Key Characteristics

We looked at the statistics obtained from the logs, and also analyzed the distribution and frequencies of messages and message tokens, and gained the following key insights:

Logs are generated at a massive rate: Table I shows that *bgl* logs were generated at a rate of 15 messages per minute and *spirit* logs at 339 messages per minute.

Logs contain large number of both alerts and informational messages: Table I shows that there were over 348 thousand and 172 million alerts in the *bgl* and *spirit* logs respectively. The logs contained many more informational messages, with approximately 4.4 million in *bgl* and 100 million in *spirit*.

Logs contain messages generated by various system components: Hardware and software components issue log messages when they encounter noteworthy events either in their own state (e.g. segmentation fault) or in their interactions with other components (e.g. failed socket read). As such, they contain information about a wide variety of phenomena that may span multiple system components.

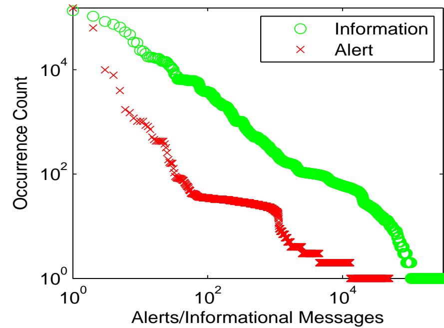


Fig. 1. Frequency distribution for the messages in the *bgl* data-set.

Logs contain redundant and duplicate information: Messages in the logs are frequently repeated over time and multiple copies of the same message could be generated by various devices. Figure 1 shows the frequency distribution for the messages in the *bgl* logs. The horizontal axis plots the individual messages in decreasing popularity order and the vertical axis shows the number of times each one occurs; both axes are logarithmic. The figure shows that informational messages are much more common than alerts. Furthermore, informational messages obey a clear power-law distribution, with a few messages that occur many times and many messages that occur a few times. Alerts follow a much more complex distribution, where common alerts follow a power-law, alerts that occur less frequently follow a linear distribution, while uncommon alerts follow a very heavy-tailed distribution. This suggests that alerts occur in complex circumstances that cannot be described using a simple statistical summary and thus require more sophisticated treatment.

Logs have unknown message-structure: While messages in the logs are generally presented as "free-text", in practice they usually represent a relatively small number of syntactic structures. To illustrate this, consider the following sample messages:

```
Error occurred in the module m1 on line 20
Error occurred in the module m2 on line 21
```

It is clear that these are different instances of the following structure:

```
Error occurred in the module <string> on
line <num>
```

TABLE I
DESCRIPTION OF THE SUPERCOMPUTING LOGS USED IN THE STUDY

Name	System	Start Date	Days	Size (GB)	Rate (lines/minute)	Messages	Alerts	Alert Categories
bgl	Blue Gene/L	2005-06-03	215	1.207	15	4,747,963	348,460	41
spirit	Spirit (ICC2)	2005-01-01	558	30.289	339	272,298,969	172,816,564	8

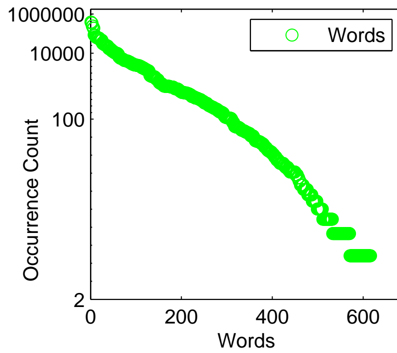
TABLE II
LOG FORMAT AND SAMPLE LOG MESSAGES

Category	Time-stamp	Date	Source	Message
-	1117838579	2005.06.03	R04-M1-N4-I:J18-U11	RAS KERNEL INFO cache parity error corrected ...
APPREAD	1117869872	2005.06.04	R23-M1-N8-I:J18-U11	RAS APP FATAL ciod: failed to read message prefix on ...
KERNDTLB	1117985502	2005.06.05	R36-M0-NC-C:J05-U01	RAS KERNEL FATAL data TLB error interrupt ...
KERNRTSP	1118073983	2005.06.06	R22-M0-N1-C:J10-U01	RAS KERNEL FATAL rts panic! - stopping execution ...
KERNSOCK	1119975388	2005.06.28	R22-M0-NC-I:J18-U11	RAS KERNEL FATAL MailboxMonitor::serviceMailboxes() ...

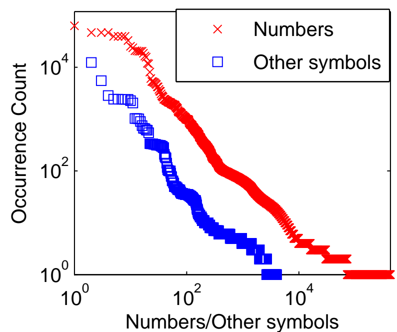
Although the set of such log structures is relatively small, it is still too large to enumerate manually. Also, because the set of system components is unknown and evolving over time, system administrators can only stay upto-date by constantly searching for new log structures.

TABLE III
DISTRIBUTION OF TOKENS IN *bgl* LOGS.

Token type	Number of unique instances
Words	685
Numbers	359051
Other symbols	126470



(a) Words have double-exponential distribution.



(b) Numbers and other symbols exhibit power-law distribution.

Fig. 2. Distribution of words in the set of unique *bgl* messages.

Log-messages contain a small set of unique words but a large number of numbers and other symbols: Table III shows the number of words, numbers and other symbols found in the *bgl* logs. It shows that while these logs contain a relatively small number of unique words and names (only 685), they have a very large number of unique numbers and other symbols. This suggests that these numbers and other symbols correspond to the large number of components, parameters and parameter settings that may exist in a system.

Distribution of words in logs is very different than that found in natural languages: Figure 2 shows the frequency distribution[1] of each unique token seen in the *bgl* dataset. The key finding is that distribution of words follows a *double-exponential* distribution (Figure 2(a)), as seen by the fact that is appears as a straight line when the horizontal axis is plotted using a linear scale and the vertical axis is plotted using a log-log scale. This distribution of words in logs is very different from the frequency distribution of words in natural languages, which are known to follow a power-law distribution [10]. This indicates that analysis techniques that are used in natural language processing may not be applicable here. On the other hand, the frequency distributions of numbers and other symbols follow power-laws. This suggests that while the log structures severely constrain the variety of words that appear in logs, numbers and other symbols are far less constrained and are thus much more diverse.

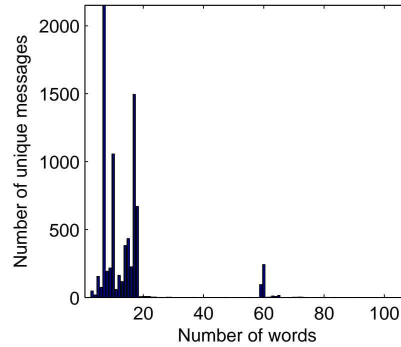


Fig. 3. Distribution of number of words per message in the *bgl* data-set.

Messages in the logs tend to be short but variable-length: Figure 3 plots distribution of number of words in the set of unique messages, showing that most contain between 5 and 20 words, although messages with as many as 100 words are also seen.

These log characteristics show that supercomputing logs are massive, and while they contain free-text messages, these actually follow underlying syntactic structures that can help to

[1]To remove the bias due to multiple instances of same messages seen in the logs, we consider the frequency distribution of tokens in the set containing unique instances of messages.

summarize log information. In the next section we discuss our textual clustering methodology to extract the underlying structure of messages. We also demonstrate how this methodology can be used in online classification of messages.

III. Extracting the Textual Structure of Logs

A. Log Preprocessing and Substitutions

The task of identifying the syntactic structure of messages is complicated by the large variability of message tokens. Specifically, while messages typically have short, consistent structures made up of a small number of words, they are parameterized by a wide variety of non-word tokens that represent entities such as port numbers, interrupt numbers and process IDs. Since these tokens can have a wide range of values, their existence significantly complicates the task of identifying the real syntactic structure of messages.

In light of this we preprocess the log to substitute non-word tokens with unique placeholder tokens. One example of this substitution is decimal and hexadecimal numbers, which are replaced with the tokens `<int>` and `<hex>`, respectively. Other examples include node ids and complex file paths such as `/etc/var/tmp1`, where "`tmp1`" is one of a large number of temporary files. For the discussion in this paper, messages after this substitution are referred to as *s-messages*.

B. Extracting Structure through Textual Clustering

Preprocessing condenses the large set of messages into a much more compact set of s-messages. We extract the true syntactic structure of s-messages by using textual clustering to group similarly-structured s-messages into different clusters. Since logs are generated incrementally over a long period of time it is important to have a clustering algorithm that can be easily adapted to work in an online fashion, incrementally producing updated clusters as new messages are seen. Furthermore, since the structure of s-messages may be complex, the clustering algorithm must be capable of tolerating noise. In light of these constraints we adapted the DBSCAN algorithm [11] for our purposes.

Our online DBSCAN algorithm works on a per-message basis. During each iteration it has 0 or more clusters and for each incoming message it computes the distance between the message and all of the existing clusters. The distance between a message and a cluster is defined as the minimum distance between the message and any message inside the cluster. If the message is sufficiently close to one or more existing clusters (its distance is below a pre-defined threshold), it is added to the nearest cluster. Otherwise, the algorithm creates a new cluster for the message and uses this cluster while processing subsequent messages. The clustering algorithm thus needs two key components: a function that defines the similarity distance between s-messages and a way to compute the threshold for message-cluster proximity. Both are discussed below.

1) Measuring Similarity: A useful distance metric for s-messages must be sensitive to their syntactic structure. As previous studies have shown [18], position of a word in a message plays an important role in identifying messages that contain alerts. Indeed, this is a key component of the manually extracted regular expressions that system administrators often use to identify alerts. This suggests that the key to identifying whether two s-messages have the same structure is a distance metric that is sensitive to the relative positions of words inside messages. Levenshtein Distance[2] [8](LD) is a well-known distance metric commonly used in text applications such as spell checking and correction, search query suggestion and information retrieval [4], [3]. It works by identifying the minimum number of operations required to transform one string into the other, where an operation may be an insertion, deletion, or substitution of a single character. Since in the case of log analysis, tokens have more semantic value than individual characters, we modified LD to use insertion, deletion or substitution of entire tokens as the operation set.

As discussed in Section II-B, messages are variable in length. Since this can cause LD to bias the clustering algorithm by giving more importance to longer strings [23], [21], we correct for this effect by dividing LD by the length of longer string. We define our Normalized Levenshtein Distance(NLD) between two messages A and B as follows:

$$NLD(A,B) = \frac{LD(A,B)}{max(length(A), length(B))},$$

where $LD(A,B)$ is the token-based Levenshtein Distance between A and B.

2) Determining the Similarity Threshold for Clustering: Figure 4 shows the distribution of NLDs between all s-message pairs in the bgl and $spirit$ logs, with NLD on the horizontal axis and the fraction of pairs that are separated by that distance on the vertical axis (log-scale). Both logs show the same basic pattern. A sizeable minority of s-message pairs are within a small distance of each other. The number of s-message pairs separated by intermediate distances is significantly smaller, with most pairs separated by large distances. This means that for every s-message, there is a clear separation between s-messages that are similar to it and those that are not. This enables us to pick the DBSCAN cluster similarity threshold by looking at the subset of pairs that are close to each other (left side of the graph) and picking the NLD value that lies in the valley immediately following this group of nearby pairs. This identifies a threshold of NLD=.15 for both the bgl and $spirit$ logs.

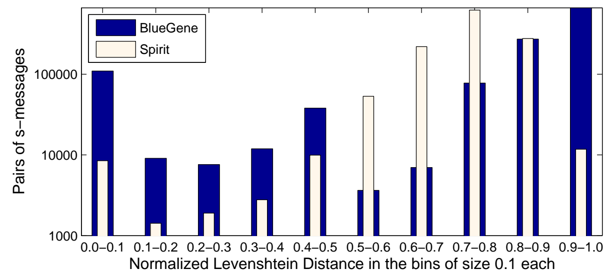


Fig. 4. Distribution of s-messages pairs based on normalized Levenshtein distance.

[2]Levenshtein Distance is also known as Edit Distance.

3) Extracting Structural Information: After performing textual clustering using the DBSCAN clustering algorithm with the similarity metrics and threshold described above, we performed an initial evaluation of the quality of the textual clustering by manually examining the clusters to ensure that each one corresponds to a well-defined syntactic structure. This examination is quite tractable since clustering the 215 days and 4.7 million messages of the bgl log results in only 616 total clusters. We extract each cluster's structure by computing the regular expression that can generate the s-messages in the cluster. Table IV shows the regular expressions for 10 representative clusters. The well-defined regular expressions generated for each of the clusters identify the syntactic structures of the corresponding log messages.

C. Online Semantic Log Classification

We now prove that the textual clusters identified by the above technique correspond to real semantic units of the logs by using them to infer the semantic meaning of messages. The bgl and $spirit$ logs have been pre-annotated by system administrators to classify messages into categories as "informational messages" or different types of alerts (41 for bgl and 8 for $spirit$). This task is labor-intensive, but provides a good way to connect messages to their semantic meaning. We extend the above textual clustering technique to automatically classify messages into semantic categories in an online manner using a minimal amount of input from the administrator. The resulting tool can be used to improve the quality of failure prediction and diagnosis and more immediately, to enhance administrators' understanding of system behavior by allowing them to easily identify parts of the system in need of attention.

Our classification algorithm extends DBSCAN to assign semantic labels to each cluster. Figure 5 outlines the overall classification process. Online DBSCAN is applied to all the messages. If a message is sufficiently close to an existing cluster, DBSCAN moves it to the cluster and assigns it the cluster's label, which may be an informational message or one of several types of alerts. If not, DBSCAN creates a new cluster that contains just that message and asks the administrator to assign this cluster a label. From that point onwards, all s-messages assigned to this newly created cluster will be assigned this label assigned by the administrator. Although we have no access to the original system administrators, we do instead have logs that have been labeled by the administrators, which we rely on as a proxy for the administrators' classification decisions. As such, when we needed to ask the administrator to label a new cluster, we instead use the label of the cluster's initial s-message.

To evaluate the performance of our online classification algorithm, we used the following metrics: (i) *Operator input:* This is the number of times the operator had to label a new cluster. This metric tries to capture the manual overhead involved in the process. (ii) *Classification accuracy:* This was measured by counting the number of times an s-message's label in the (annotated) log matches the label of its cluster. If the labels matched, the s-message was considered to be classified correctly, otherwise, it was considered to be misclassified.

Figure 6 shows the performance of the classification algorithm on the bgl logs (entire log) and $spirit$ logs (initial 108 days) relative to the cluster similarity threshold (described above). As the threshold grows larger, the number of clusters generated by the algorithm, and hence, the operator input, falls significantly as is becomes easier and easier to connect each incoming message into an existing cluster. However, at the same time, larger threshold values result in poorer classification performance. In particular Figure 6 shows perfect classification for threshold values upto .1 on bgl and .2 on $spirit$, growing worse with more relaxed thresholds. This validates our use of average distances between message pairs from Figure 4 to choose a good threshold since these choices feature few mis-classifications in reality.

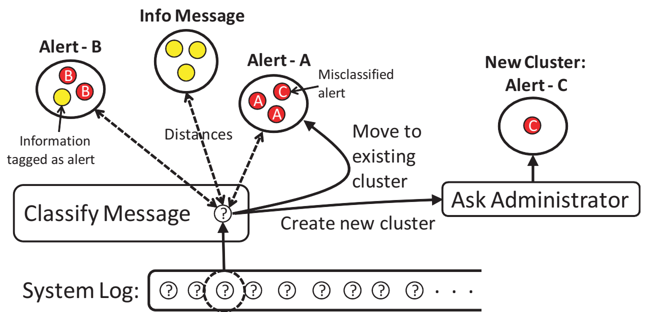


Fig. 5. Overview of classification methodology

We now explore the detailed performance results of the classification algorithm, specifically focusing on the bgl logs with threshold=.15. Figure 7 shows how the total number of clusters increases as the algorithm operates on the log's full 215 days of messages (this is also the number of classification decisions the algorithm needs from the administrator). To classify bgl's 4.7 million messages the algorithm requires 616 administrator decisions, approximately 200 of which occur during the initial learning phase. After this the rate of decisions remains relatively steady for the remainder of the log at approximately 2 decisions per day. If desired, this number can be reduced even further by changing the similarity threshold to reduce the load on the administrator at the cost of more misclassifications. Furthermore, since multiple systems are likely to execute related software and face similar alerts, it is possible to significantly reduce administrator work by sharing decisions across multiple related systems.

A key limitation of the online DBSCAN algorithm is that as it analyzes more messages, it needs to use more and more of them to decide whether a given incoming message belongs in some existing cluster. Specifically, while it took <10ms to classify messages early on in the log, this time increased to 1 second per message after classifying 4.6M lines of log messages. This was due to more comparisons required later on as the set of s-messages increased with time. We used two techniques to improve the performance of clustering. First, we

Cluster id	Message-structure
1	KERNEL FATAL disable all access to cache directory <int>
2	KERNEL FATAL rts:kernel terminated for reason <int>
3	KERNEL INFO <int>ddr errors(s)detected and corrected on rank <int>,symbol <int>,bit <int>
4	APP FATAL ciod:LOGIN chdir(<token,*>)failed:No such file or directory
5	DISCOVERY ERROR Found invalid node ecid in processor card slot <token,J*>,ecid <EC id>
6	DISCOVERY INFO Node card VPD check:<token,U*1>node in processor card slot <token,J*>do not match.VPD ecid <EC id>,found <EC id>
7	DISCOVERY SEVERE Problem communicating with service card,ido chip:<hex pair>:<hex pair>:<hex pair>.java.io.IOException:Could not find EthernetSwitch on port:address <hex pair>
8	APP FATAL ciod:Error reading message prefix after <token,*>on CioStream socket to <int>.<int>.<int>.<hex pair>:Link has been severed
9	CMCS INFO Controlling BG/L rows [<int><int><int><int><int><int><int><int>]
10	MMCS INFO idoproxydb has been started:$Name:<token,V1R1*>$ Input parameters:-enableflush -loguserinfo db.properties BlueGene1

TABLE IV
MESSAGE-STRUCTURES REPRESENTED BY SOME SAMPLE TEXTUAL CLUSTERS EXTRACTED FROM *bgl* DATASET

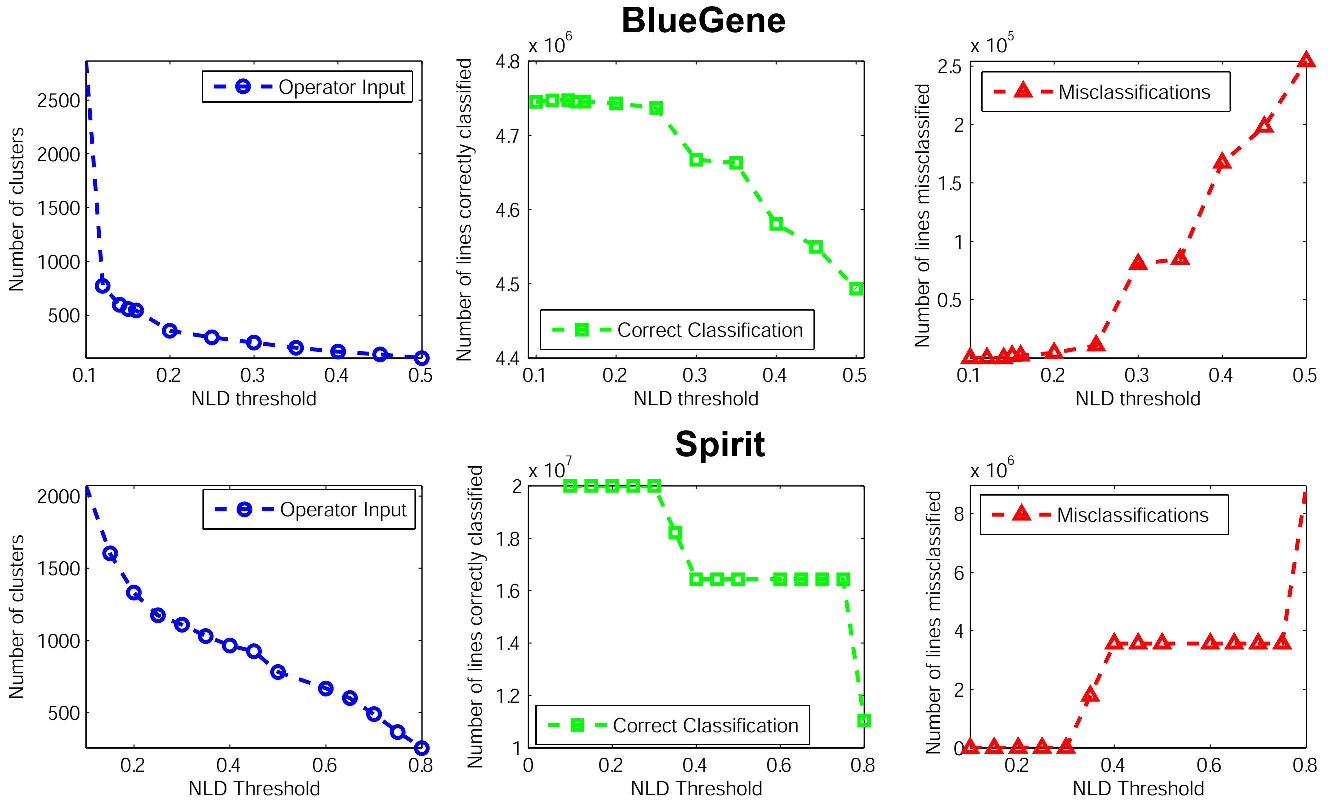


Fig. 6. Performance of message classification on the *bgl* and *spirit* logs

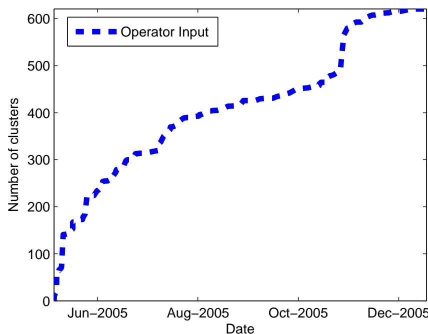


Fig. 7. Operator input for tagging newly seen messages with time

detect duplicate s-messages and only classify each unique s-message once, which removes the additional comparison costs and reduces the total number of messages in the clusters. Further, we impose an upper bound on the size of any cluster, removing each cluster's oldest message when the cluster reached this bound. Figure 8 shows how online DBSCAN performs after duplicate message elimination as the upper bound k ranges from 1 to 64 to no bound. Figure 8(a) shows that average time spent per message drops dramatically from 343ms per message with no upper bound on cluster size to just 47ms when cluster size is bounded. Figure 8(b) explains the reason, showing how the total number of messages considered by DBSCAN rises much more slowly when cluster upper bounds are imposed. Looking at classification performance, Figure 8(c) shows that even as the maximum cluster size ranges from 1 to no bound, the rate of mis-classifications

does not change at all, demonstrating the robustness of the approach. Indeed, as Figure 8(d) illustrates, the only reason to use limits larger than 1 is to reduce the number of clusters created, which drop from 616 to 578, with k between 4 and 16 being good choices from the perspective of higher processing throughput, good classification performance and small overhead for the administrator.

Finally, Table V summarizes the ultimate effectiveness of the message classification algorithm on both the bgl and $spirit$ logs, showing that our online classification algorithm has a near perfect classifications rate for bgl (More than 99%), and perfect classification rate for $spirit$. A total of 4364 messages (including duplicates) are mis-classified for bgl logs. However, these 4364 messages are different instances of only 5 unique s-messages, which can be further broken into 2 information s-messages being tagged as alert and 3 alert s-message being tagged with wrong alert labels.

	BlueGene	**Spirit**
Number of days	215	109
Total number of log lines classified	4.7M	31M
Number of mis-classified non-unique s-messages	4364	0
Number of mis-classified unique s-messages	5	0
Unique alert s-messages tagged as information	0	0
Unique information s-messages tagged as alert	2	0
Unique alert s-messages tagged as wrong alert	3	0

TABLE V
COMBINED MIS-CLASSIFICATION RATES FOR bgl AND $spirit$ LOGS

IV. TEMPORAL ANALYSIS OF LOGS

Extracting log structure using textual clustering enables us to organize the large volume of logs at the level of individual lines. Unfortunately, this approach cannot on its own provide insight into larger correlations across different events occurring in the system. Such correlations can provide a better understanding of interactions among different components and become vital in identifying the causes of system failures and performance bottlenecks. For example, the failure of a network card may cause a variety of network applications to throw error messages. However, since these applications produce very different messages, the only way to identify their common cause is to detect the temporal correlation between them.

In this section, we describe a way to automatically detect such correlations for various textual clusters. The basic idea behind our technique is to perform temporal clustering of textual clusters based on the likelihood that different textual clusters appear at the same time. In addition to identifying textual clusters that may be causally related, this technique can extract interesting cross-application dependency patterns such as different applications reacting to the unavailability of a common resource. Furthermore, although textual clustering only considers individual log lines, temporal clustering can also identify multi-line messages, where all the individual lines always appear together.

A. Constructing Textual Cluster Time-Series

Temporal clustering of textual clusters works by dividing time into multiple discrete time-slots and counting the frequency of occurrence of each textual cluster appearing in a given time-slot. Since textual clusters consist of many individual s-messages, for each time-slot we increment the cluster's counter every time a member s-message appears in the time-slot. We say that a given textual cluster "is seen" during a given time-slot if its counter is greater than 0.

Figure 9(a) shows the time-series for a sample data-set taken from bgl logs. The vertical axis represents different textual clusters and horizontal axis represents consecutive time-slots. Each row thus represents a time series for a textual cluster.

B. Time-Series Clustering to Extract Temporal Structure

Figure 9 shows that although the distribution of textual clusters over time is relatively sparse, there seem to be correlated occurrences over time across sets of clusters. We extract this temporal structure by clustering textual clusters into larger groups of correlated clusters using Agglomerative Hierarchical Clustering(AHC).

AHC starts with all textual clusters broken up into separate temporal clusters and merges the two closest clusters successively until it reaches the target number of clusters. Cluster distance between temporal clusters C_1 and C_2 is defined as the average distance between all pairs of textual clusters in C_1 and C_2, which has been shown to be robust against noise and outliers in the data-set [20]. The process of cluster aggregation stops when the target number of clusters is reached. We now define the distance function we use with AHC and the algorithm for picking the optimal number of clusters.

To reliably identify the temporal correlations between two textual clusters, a distance metric must rely only on time-slots when either cluster is actually seen and ignore the rest. This is because the time series are sparse (Figure 9(a)), and any two textual clusters appear in only a small fraction of the overall range of time-slots. We thus use the "Jaccard-distance" metric, which is the ratio of number of time-slots when only one of the two textual clusters is seen and the number of time-slots in which either one or both are seen. This metric thus measures the likelihood of not seeing two textual-clusters together, with larger Jaccard distance implying a lower likelihood of the two clusters being correlated.

Formally, consider the time series T_i and T_j for two textual-clusters i and j respectively, and denote the number of time-slots where neither T_i nor T_j are seen as M_{00}, the number of time-slots in which only T_i (T_j) is seen as M_{01} (M_{10}), and the number of time-slots where both are seen as M_{11}; then Jaccard distance between T_i and T_j denoted by $J_\delta(T_i, T_j)$ is defined as:

$$J_\delta(T_i, T_j) = \frac{M_{01} + M_{10}}{M_{01} + M_{10} + M_{11}}$$

Figure 9(b) shows pair-wise Jaccard distances for the time-series of textual clusters in the same sample as in Figure 9(a).

In general it is not possible to identify the correct number of temporally correlated clusters in the logs before temporal clustering is performed. As such, our algorithm for picking the optimal number of clusters works by looking at the quality

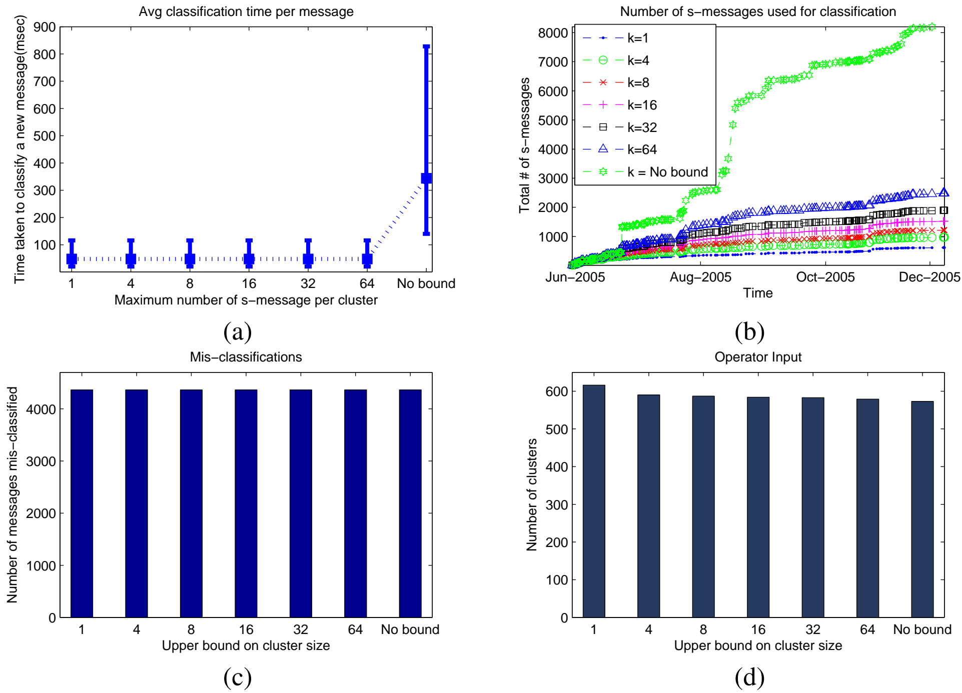
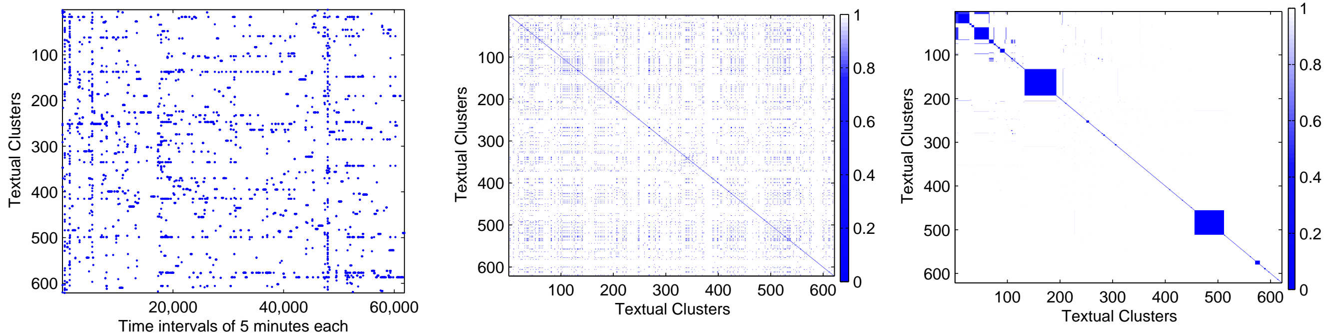


Fig. 8. Results with varying cluster-size upper bounds



(a) Binary time series for individual textual clusters; a dot represents the fact that at least one member of a textual cluster was seen in the corresponding time-slot.

(b) Jaccard distance for the pairs of textual-clusters.

(c) Jaccard distance for the pairs of textual-clusters after reordering based on clustering.

Fig. 9. Temporal analysis on textual-clusters for *bgl* logs.

of the clusters generated during different iterations of AHC and choosing the smallest cluster count that optimizes this quality. We use the average weighted intra-cluster distance $WIntraCD$ as an estimate of the quality of a given clustering. $WIntraCD(C_i)$ is defined as the mean of all the pair-wise distances between the elements of cluster C_i, multiplied by C_i's size. Intra-cluster distance is a good estimate of cluster quality since in good clustering all members of all clusters will be near each other. The weighting ensures that a given well- or badly-placed textual cluster will be equally important regardless of whether it is in a large temporal cluster or a small one. Further, it ensures that the average $WIntraCD$ increases monotonically as AHC merges nearby clusters, which makes it easier to choose the correct number of clusters. Figure 10 plots the average $WIntraCD$ as AHC progresses from over 600 clusters to 1 (looking from right to left) on the sample from Figure 9. There is a clear point at 400 temporal clusters where $WIntraCD$ begins to rise as textual clusters that correlate poorly with each other begin to be forcibly fused by AHC into the same temporal cluster. We thus stop AHC at this knee in the $WIntraCD$ graph since it corresponds to the minimum number of high quality clusters.

Figure 9(c) plots the pair-wise distances for the time-series of textual clusters shown in Figure 9, except that the textual clusters have been reordered to group together ones that fall into the same temporal cluster. This plot now shows several solid blocks that correspond to groups of textual clusters that are all temporally correlated to each other (these textual clusters consistently appear in the same time-slots) and are not correlated to other textual clusters (either do not appear together or appear together inconsistently). This suggests that there exists significant temporal correlation across different textual clusters that can be extracted using our temporal clustering methodology.

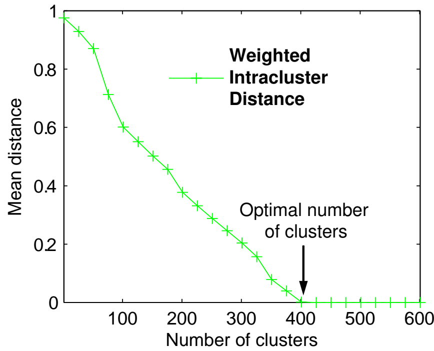


Fig. 10. Intra cluster distance for temporal clusters

TABLE VI
CLASSIFICATION OF TEMPORAL CLUSTERS

Temporal Cluster Type	Count
Node card related event clusters	6
Node start/shutdown	4
Possible repair by system administrator	4
Application events	4
Log-in events	2
Unknown events	3
Temporal clusters with just one textual-cluster	370
Multi-line log messages	22

C. Finding Semantics from Temporal Correlation

Temporal clusters provide us with useful information about various events happening in the system. Using our methodology, we extracted all the temporal clusters for the *bgl* dataset. To understand what these clusters mean, we carefully examined each temporal cluster and classified them into different categories of events. Table VI shows the results of this classification. The first insight is that our methodology extracts a significant number of correlated textual-clusters from the *bgl* logs. These include 22 multi-line log messages that were broken up into multiple single-line messages (some clusters contain more than one multi-line message). Further, 20 temporal clusters contain messages that all correspond to the same system event such as a node starting up or shutting down (some multi-line messages may also fall into one of these categories). Table VII provides several examples of these clusters. This information will not only be useful in extracting the correlated failures with their possible causes, but can also help improve system management by identifying performance bottlenecks and cross-application dependencies. Finally, 370 temporal clusters only contain a single element, suggesting that these textual clusters do not have a clear correlation with other clusters. The study of additional temporal relationships between these messages is part of our ongoing work.

V. RELATED WORK

Most of the previous work on extracting information from supercomputing logs has mainly relied on techniques based on regular expressions [1], [7], [14]. However, creation and maintenance of these regular expressions requires laborious and active examination of log messages. Also, these logs change with time due to various system upgrades and new applications, making it painstakingly difficult for human experts to maintain these regular expressions.

To utilize the information contained in logs several researchers have proposed the idea of using better visual aids. Tudumi is an information visualization system proposed for monitoring and auditing computer logs [19]. Though it is interesting to visualize these things, still these tool are limited in their capabilities and can not be used to extract detailed information from the logs such possible event correlations, textual structure of the logs, classification of logs into various categories etc.

Various studies have looked at understanding logs and their usefulness for detecting faults in supercomputers [18], [13], [17]. These studies show many interesting characteristics of these logs. For example, Stearley et. al. [18] showed that though words alone do not help in detecting alert messages from logs, words coupled with their position information can be a powerful indicator for differentiating alert messages. This key finding suggests that log messages are likely to contain structure that can separate log messages into semantic categories. Our work has verified this basic finding, providing significant additional detail and demonstrating how to do this in an automated and online manner.

Researchers have also looked at utilizing logs other than system logs, such as application console logs [22]. However, this technique is limited to application specific anomalies and requires source code, which may not be available for proprietary systems. In another related work Salfner and Tschirpke used Levenshtein Distance to preprocess error messages in a commercial telecommunication system [15], before passing them into a failure prediction tool.

VI. CONCLUSION

The large size and complexity of supercomputers makes them difficult to manage. Containing hundreds to tens of thousands of nodes, Terabytes of memory, Petabytes of storage and complex network topologies, these systems can exhibit a very wide variety of complex behaviors that result in performance degradation and component failure. System logs are commonly used by operating systems to record important informational and alert messages from various software and hardware components such as daemons, applications and drivers. Because they contain key status details about the health of many system components, they represent an invaluable resource for understanding the status of large systems, which can help make them easier to manage. However, their poor semantics and the overwhelmingly large size of the combined logs of all supercomputer nodes severely limits the usefulness of these system logs to system administrators.

This paper presents a foundational study on identifying the structure of the basic units of system logs: messages from system components. We examined various properties of log messages including the distribution of the individual message terms and structures, identifying previously unknown statistical properties such as the fact that message words follow a double-exponential distribution rather than the expected

TABLE VII

EXAMPLES OF SOME OF THE TEMPORALLY CORRELATED LOG-STRUCTURES EXTRACTED FOR *bgl* LOGS

Cluster id	Temporally-correlated log-structures
1	**A node card failure event**
	LINKCARD INFO MidplaneSwitchController performing bit sparing on R25-M1-L0-U22 - <hex>bit <num>
	HARDWARE SEVERE LinkCard power module U58 is not accessible
	DISCOVERY SEVERE Problem communicating with service card,ido chip: . . .
	java.io.IOException:Could not find EthernetSwitch on port:address . . .
	HARDWARE WARNING PrepareForService is being done on this part(mLctn(R73-M1-N7),mCardSernum(<hex>),. . . by root
2	**Multi-line, correlated due to shared interrupt**
	KERNEL FATAL auxiliary processor <int>
	KERNEL FATAL byte ordering exception <int>
	KERNEL FATAL data store interrupt caused by <word,*><int>
	KERNEL FATAL program interrupt: <word,*>..<int>
	KERNEL FATAL store operation <int>
	KERNEL FATAL program interrupt: fp cr <word,*><int>
	KERNEL FATAL exception syndrome register: <hex>
	KERNEL FATAL machine check: i-fetch <int>
3	**Restart of a module**
	MMCS INFO idoproxydb has been started: $Name: <word,V1R1M0*>$ Input parameters: -enableflush -loguserinfo db.properties BlueGene1
	MMCS INFO ciodb has been restarted.
	BGLMASTER INFO BGLMaster has been started: ./BGLMaster –consoleip <ip add>–consoleport <int>–configfile bglmaster.init –autorestart y
	MMCS INFO mmcs_db_server has been started: ./mmcs_db_server –useDatabase BGL –dbproperties serverdb.properties –iolog /bgl/BlueLight/logs/BGL –reconnect-blocks all –shutdown-timeout <int>
4	**Possibly a multi-line message**
	CMCS INFO Running as background command
	CMCS INFO Controlling BG/L rows [<int><int><int><int><int><int><int><int>]
	CMCS INFO Starting SystemController

power-law. We analyzed the syntactic structure of messages by developing a novel textual clustering algorithm that groups messages according to their structures. We then showed that the discovered structures were the semantically meaningful units of system logs by developing an online message classification algorithm that can accurately replicate the message annotation decisions made manually by system administrators. Finally, we used temporal clustering to identify correlations between message occurrence times, discovering new features in the log, including complex events and multi-line messages.

The fundamental contribution of this work is to identify the basic informational units that system logs are composed of. Our techniques thus enable a wide variety of future sophisticated system log analysis, providing them with log entries that have significantly richer semantic properties than traditional textual log lines. Our two example analyzes, alert classification and temporal clustering, prove the basic usefulness of our approach and point the way to future techniques that will further leverage this work.

REFERENCES

[1] Logsurfer and logsurfer+- real time log monitoring and alerting. http://www.crypt.gen.nz/logsurfer/.

[2] Supercomputer event logs from sandia national laboratories. http://www.cs.sandia.gov/~jrstear/logs/.

[3] F. Ahmad and G. Kondrak. Learning a spelling error model from search query logs. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 955–962. Association for Computational Linguistics Morristown, NJ, USA, 2005.

[4] S. Cucerzan and E. Brill. Spelling correction as an iterative process that exploits the collective knowledge of web users. In *Proceedings of EMNLP*, volume 4, pages 293–300, 2004.

[5] John Daly. Running applications successfully at extreme scale: What is needed? In *ASCR Computer Science Research Principal Investigators Meeting*, 2008.

[6] John T. Daly. Performance challenges for extreme scale computing. In *SDI Seminar, Carnegie Mellon University*, 2007.

[7] S.E. Hansen and E.T. Atkins. Automated System Monitoring and Notification With Swatch. *In USENIX LISA'93 Conference Proceedings*, 1993.

[8] VI Levenshten. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics-Doklady*, volume 10, 1966.

[9] C. Lonvick. The BSD syslog protocol, 2001.

[10] C.D. Manning and H. Schütze. *Foundations of statistical natural language processing*. MIT Press, 1999.

[11] Jrg Sander Xiaowei Xu Martin Ester, Hans-Peter Kriegel. A density-based algorithm for discovering clusters in large spatial databases with noise. In *International Conference on Knowledge Discovery and Data Mining*, pages 226–231, 1996.

[12] Mark Megerian. BG/L control system software. http://www.mcs.anl.gov/ beckman/bluegene/SSW-Utah-2005/BGL-SSW07-ControlSys.pdf, 2005.

[13] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 575–584, June 2007.

[14] J.E. Prewett and E. James. Listening to Your Cluster with LoGS. In *The Fifth LCI International Conference on Linux Clusters: TheHPC Revolution 2004*, 2004.

[15] F. Salfner and S. Tschirpke. Error Log Processing for Accurate Failure Prediction. In *Sixth International Conference on Information Visualisation.*, pages 570–576, 2002.

[16] B. Schroeder and GA Gibson. A large-scale study of failures in high-performance computing systems. pages 249–258, 2006.

[17] J. Stearley. Towards informatic analysis of syslogs. In *IEEE International Conference on Cluster Computing*, pages 309–318, 2004.

[18] J. Stearley and A.J. Oliner. Bad words: Finding faults in Spirit's syslogs. In *CCGRID'08, 8th IEEE International Symposium on Cluster Computing and the Grid*, pages 765–770, 2008.

[19] T. Takada and H. Koike. Tudumi: Information visualization system for monitoring and auditing computer logs. In *Information Visualisation, 2002. Proceedings. Sixth International Conference on*, pages 570–576, 2002.

[20] P.N. Tan, M. Steinbach, and V. Kumar. *Introduction to data mining*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2005.

[21] F. Weigel, A. Fein. Normalizing the weighted edit distance. In *Pattern Recognition, 1994. Vol. 2 - Conference B: Computer Vision & Image Processing., Proceedings of the 12th IAPR International. Conference on*, pages 399–402, Jerusalem, Israel, 1994.

[22] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Mining console logs for large-scale system problem detection. In *Workshop on Tackling Computer Problems with Machine Learning Techniques (SysML), San Diego, CA*, 2008.

[23] Li Yujian and Liu Bo. A normalized levenshtein distance metric. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(6):1091–1095, 2007.