Detecting Abnormal Machine Characteristics in Cloud Infrastructures

Kanishka Bhaduri
MCT Inc., NASA Ames
Moffett Field, CA-94035
Kanishka.Bhaduri-1@nasa.gov

Kamalika Das
SGT Inc., NASA Ames
Moffett Field, CA-94035
Kamalika.Das@nasa.gov

Bryan L. Matthews
SGT Inc., NASA Ames
Moffett Field, CA-94035
Bryan.L.Matthews@nasa.gov

***Abstract*—In the cloud computing environment resources are accessed as services rather than as a product. Monitoring this system for performance is crucial because of typical pay-per-use packages bought by the users for their jobs. With the huge number of machines currently in the cloud system, it is often extremely difficult for system administrators to keep track of all machines using distributed monitoring programs such as Ganglia[1] which lacks system health assessment and summarization capabilities. To overcome this problem, we propose a technique for automated anomaly detection using machine performance data in the cloud. Our algorithm is entirely distributed and runs locally on each computing machine on the cloud in order to rank the machines in order of their anomalous behavior for given jobs. There is no need to centralize any of the performance data for the analysis and at the end of the analysis, our algorithm generates error reports, thereby allowing the system administrators to take corrective actions. Experiments performed on real data sets collected for different jobs validate the fact that our algorithm has a low overhead for tracking anomalous machines in a cloud infrastructure.**

I. Introduction

Cloud Computing [1] refers to the infrastructure in which applications are delivered as services over the Internet. These infrastructures are supported by very large networked distributed machines. Users typically can pay for the time they would like to use these resources, *e.g.* CPU usage per hour or storage costs per day. This mode of computation is beneficial to both the user and provider for several reasons:

- by allowing pay-per-use model, the users can run their jobs with less cost investment compared to owning the machines themselves
- since there is a cost associated with the loan of the resources, users will always have an incentive to return them, when no longer needed
- an easy way for the cloud provider to add resources when the demands are not met anymore

With the introduction of any new technology, there is always a need for developing techniques for health assessment of these systems. This is true even in the case of clouds, where providers strive for availability and responsiveness since expectations on the side of the users are high. System administrators in charge of such systems have a daunting task in maintaining them given hundreds and thousands of machines in the system. In case of failures, these faults may quickly propagate causing wide spread damage. Therefore system administrators would like to automatically detect these faults as early as possible for early mitigation strategies.

Event monitoring programs such as Ganglia provides a web based visualization interface for allowing the system administrators to view different parameters pertaining to the health of each of the machines in the distributed infrastructure. Detailed list of the parameters are given in Section V. Given there are hundreds to thousands of machines in the system, visual inspection of system performance may be too late or nearly impossible. Moreover, it is also imperative to isolate the fault to a few subset of variables (fault isolation). An automated fault detection and isolation technique is necessary in such scenarios.

In this paper, we describe an automated fault detection framework for cloud system *FDCS* which runs on top of the Ganglia system. The algorithm is entirely decentralized; as a result does not burden any single machine with excessive workload and at the same time does not require all the data to be centralized for execution. *FDCS* takes all the measurements of Ganglia into consideration and reports a ranked list of the machines based on its anomaly or fault score. Moreover, for each machine in this list, a system administrator can display the most faulty variable which caused the anomaly. The algorithm uses distance based anomaly definition to identify if a machine is faulty or not. It is extremely fast and can run continuously on changing data, thereby allowing an uninterrupted monitoring of the machine performance. Using *FDCS*, one can take corrective actions early before they become fatal faults and thereby degrading the overall system performance.

The rest of the paper is organized as follows. In Section II we discuss some previous work related to this area of research. Next in Section III we discuss the notations and problem definition. In Section IV we present our fault detection and isolation (*FDCS*) framework. Empirical evaluation is presented in Section V. Finally, we conclude and discuss some future directions in Section VI.

II. Related Work

In this section we present some work related to this area of research.

[1] ganglia.sourceforge.net/

Arshad *et al.* [2] presents a framework for intrusion detection and diagnosis for clouds. The goal of the paper was to map the input call sequences to one of the five severity levels: "minimal", "medium", "serious", "critical", and "urgent". The authors have used decision trees for this task. The tree learns rules which can perform predictions on unseen instances. Experiments with publicly available system call sequences from the University of New Mexico (UNM) show that the algorithm exhibits good performance. A similar approach was also developed by Zheng *et al.* [3] and [4]. The last paper uses canonical correlation analysis (CCA) for tracking maximally correlated subspaces over time. One problem with both these techniques is that they both need labeled examples for training which are difficult to acquire.

Most of the existing techniques for failure detection are rule-based [5] which defines a set of watchdogs. The method comprises of monitoring a single sensor using some hard thresholds. Whenever, the sensor value crosses the threshold, an alarm is raised. However, this threshold needs to be changed for different types of jobs to prevent missed detections and false alarms.

Bodik *et al.* [6] develop a method for identifying time cycles in machine performance which fall below a certain threshold. They use quantiles of the measured data to statistically quantify faults. They optimize the false positive rate and provide the user to directly control it. This method was evaluated on a real datacenter running enterprise level services giving around 80% detection accuracy. However, as with some of the previous techniques, this method too requires labeled examples. An overview article on this topic is available at [7].

Pelleg *et al.* [8] explore failure detection in virtual machines. They use decision trees to monitor counters of the system. First of all, this method requires labeled instances for training and. Moreover, the counters which are monitored are manually detected which reduces the scope of its general applicability. It is only suitable for well managed settings that include predictable workloads and previously seen failures.

Some data mining techniques have also been applied for monitoring distributed systems *e.g.* the Grid Monitoring System (GMS) by Palatin *et al.* [9] and the fast outlier detection by Bhaduri *et al.* [10]. GMS uses a distributed distance-based outlier detection algorithm, which detects outliers using the average distance to k nearest neighbors. Similar to our method, GMS is based on outlier detection and is unsupervised and requires no domain knowledge. But the detection rate of GMS can be very slow due to the quadratic time complexity of k-nn computation. The authors in [10] propose to speed up this computation using fast database indexing and distributed computation.

Gabel *et al.* [11] presents a technique for latent fault detection on clouds. The proposed framework is unsupervised and based on statistical tests for fault detection. The main idea behind it is to compare machines performing the same task at the same time. A machine is flagged as abnormal when it deviates from the normal behavior. The authors demonstrate three tests within this framework and provide theoretical guarantees on the false detection rates of the proposed tests. The experiments are performed on several production services of various sizes and natures, including ones using virtual machines. However, this method is not distributed, thereby requiring one machine to run the tests.

III. Notations and Problem Definition

In this section we present some notations which are necessary for discussing our *FDCS* framework.

Let $P_1, \ldots, P_p$ be p machines in the cloud infrastructure connected to each other via a communication infrastructure such that the set of (one-hop) neighbors of P_i, Γ_i is known to P_i. Each P_i holds a dataset D_i (*e.g.* its status or log file) containing n vectors each in $\mathbb{R}^d$. We assume

- **Disjoint property:** $D_i \cap D_j = \emptyset, \forall i \neq j$
- **Global property:** $D = \bigcup_{i=1}^{p} D_i$

In real applications, it is not feasible to compute D due to massive data sizes, changing datasets or both. In this paper, we have only introduced this notation to formally define our global fault detection task via distributed processing.

Given two user-defined parameters $t, k > 0$, let $N_k(x, D)$ denote the set of k nearest neighbors from $\{D \setminus \{x\}\}$ to x (with respect to Euclidean distance with ties broken according to some arbitrary but fixed total ordering $\prec$). Let $\delta_k(x, D)$ denote the maximum distance between x and all the points in $N_k(x, D)$ *i.e.* the distance between x and its k-nearest neighbors in D. $\delta_k(x, D)$ can be viewed as an *outlier ranking function*. Let $O_{t,k}(D)$ denote the top t points (outliers) in D according to $\delta_k(., D)$. In the rest of the description, for simplicity, we rewrite $N_k(b, D)$, $\delta_k(b, D)$ and $O_{t,k}(D)$ as $N_k(b)$, $\delta_k(b)$ and O_k.

Definition 3.1 (Distributed fault detection): Given integers $t, k > 0$, and dataset D_i at each machine P_i, the goal of distributed fault detection algorithm is to compute the outliers O_k (in $D = \bigcup D_i$).

In the above definition, we have assumed that the distributed outlier detection algorithm produces the same set of outliers as its centralized counterpart [12]. The distributed algorithm that we discuss in this paper guarantee global correctness.

IV. Fault Detection in Cloud Systems (FDCS)

In this section we describe our Fault Detection in Cloud Systems (*FDCS*) framework in which the participating machines in a cloud computing environment can collaboratively track the performance of other machines in the system and raise an alarm in case of faults. Our algorithm relies

on in-network processing of messages, thereby making it faster than the brute force alternative approach of data centralization. Moreover, as we discuss in this section, it also allows fault isolation — determining which features are most faulty — which is valuable to take remedial actions.

In our distributed setup, we assume that there is a central machine in the cloud infrastructure called reporter which does the final reporting of all the outliers. We also assume that all computational entities $P_1, \ldots, P_p$ form a unidirectional communication ring (except the leader machine P_0) *i.e.* any machine P_i can communicate with the machine with the higher id $P_i + 1$, $1 \leq i < p$. Furthermore, each machine holds its own data partition D_i while the test points are either sent by P_0 or read from the disk.

At any point of time, P_0 maintains a current list of t outliers O_k found so far. These are the points which, by definition, currently have the highest anomaly scores $\delta_k(, D)$ on the global dataset D. When the algorithm starts, O_k is empty and it gets updated as new candidate outliers are received from $P_1, \ldots, P_p$. Another quantity which the reporter needs to maintain is the cutoff threshold c which is initially set to $-\infty$ and it monotonically increases in value as more and more outliers are found. Whenever, O_k changes, it is set to the smallest value in O_k and then broadcast to all the other machines in the cloud for more efficient pruning of outlier points.

Algorithm 1: *FDCS* push mode at any machine P_i.

Procedure PUSH_Anom()
begin
  for *all blocks of data in* D_i **do**
    $B \leftarrow \mathbf{getNextBlock}(D_i)$;
    for *all points* $b \in B$ **do**
      $\mathcal{L}_k(b) \leftarrow \emptyset$;
    for *all points* $x \in D_i$ **do**
      for $b \in B, b \neq x$ **do**
        if *dist*$(b, x) < r_b$ *or* $|\mathcal{L}_k(b)| < k$ **then**
          Update $\mathcal{L}_k(b)$ with x by removing the farthest point;
          Recompute r_b;
          if $r_b < c_i$ **then**
            remove b from B;
            $\tau_i \leftarrow \tau_i + 1$;
    for $b \in B$ **do**
      Send $(b, \mathcal{L}_k(b), r_b)$ to machine P_{i+1} mod p;
    Call PULL_Anom();

In *FDCS*, each worker has two modes of operation *push* and *pull*. Alg. 1 gives the pseudo code for the push mode.

Algorithm 2: *FDCS* pull mode at any machine P_i.

Procedure PULL_Anom()
begin
  for *all* $x \in$ *received buffer* **do**
    Extract $(x, \mathcal{NN}_k(x), r_x)$ from *received buffer*;
    Update $\mathcal{L}_k(x)$ using $N_k(x)$ and $\mathcal{NN}_k(x)$;
    Update r_x;
    if $r_x > c_i$ **then**
      if x *originated in machine* P_i **then**
        Send $(x, r_x, \mathcal{L}_k(x))$ (a potential outlier message) to the reporter machine (P_0);
      else Send $(x, \mathcal{L}_k(x), r_x)$ to machine P_{i+1} mod p;
    else $\tau_i \leftarrow \tau_i + 1$;

The goal of the push mode is to test a block of data read from the memory, populate its k-nn based on its local dataset, prune the points which are less than the current threshold c_i and then send the residual number of test points to the next machine in the ring. The details of this step are as follows. Machine P_i maintains a threshold c_i it has received from the reporter P_0. Initially $c_i = -\infty$. For each point b in the test data block B, machine P_i also maintains:

- $\mathcal{L}_k(b)$— the k-nearest neighbors found thus far for b
- $r_b = \max\{\|b - y\| : y \in \mathcal{L}_k(b)\}$

Initially, $\mathcal{L}_k(b) \leftarrow \emptyset$ and $r_b = 0$ for each point $b \in B$. The algorithm populates $\mathcal{L}_k(b)$ for b and checks to see if the current score of b is below c_i *i.e.* if $r_b < c_i$. If this is true, then the point is no longer tested and pruned; otherwise b along with its nearest neighbors found so far $\mathcal{L}_k(b)$ and r_b are forwarded (*pushed*) to the next machine P_{i+1} for validation.

In the second phase of *FDCS*, which is the *pull* phase, the goal of the algorithm is to check the received buffer for messages, extract the anomalies and their nearest neighbors and merge the nearest neighbors with the existing ones. The pseudo code is shown in Alg. 2. For every point x in the received buffer, P_i finds the nearest neighbors from $\mathcal{NN}_k(x)$ (which are the best set of k neighbors found so far) and D_i. The neighbor list and the value of r_x are updated accordingly. As a result, if r_x becomes less than c_i, then x is pruned. Otherwise, if x originated in P_i itself, it has survived the pruning of all the machines and is sent to the leader machine P_0 (since it can be a potential outlier data point). If x did not originate on P_i, is forwarded to P_{i+1} with the updated nearest neighbors. Machine P_i then goes back to the *push* mode and begins testing the new set of points. In any step of the execution, if any machine gets a new cutoff threshold c, it immediately sets $c_i \leftarrow c$ and resumes the processing.

Alg. 3 shows the tasks executed by the leader machine in

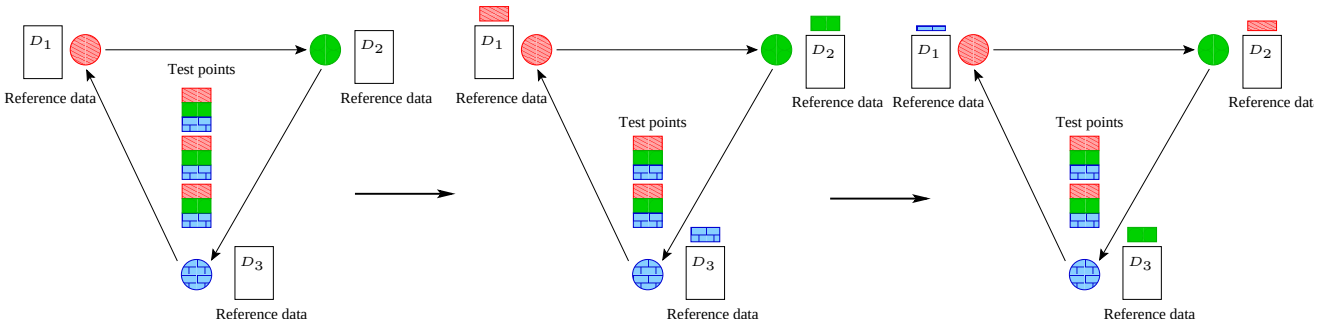


Figure 1. Execution of distributed algorithm. The leftmost picture shows the setup: the test points are color coded to show which block is assigned to which machine. Second picture shows that assignment. Third figure shows how the non-pruned points are tested at the other machines.

FDCS. It initializes the outlier list O_k to null. Whenever it receives a new potential outlier it does one of the following:

- If O_k contains less than t outliers, x is added to O_k
- If O_k contains t outliers, the outlier already in O_k with the smallest score is replaced by x.

If due to either of these computations, the outlier list becomes full, the cutoff is set to the score of the smallest outlier and it is then broadcast to all the machines.

Fig. 1 shows a snapshot of the distributed algorithm. The leftmost figure shows how 3 machines are connected in a ring. The test points are shown in the middle, color coded to show that each block is assigned to one machine. The second figure shows its initial assignment. As the test blocks move in the ring, each machine prunes points as nearest neighbors are found. As a result, the size of the test blocks shrinks. This is shown in the last figure.

One critical component of any distributed algorithm is the termination criterion. In *FDCS* this can be implemented in one of two ways. Each machine P_i keeps track of τ_i, the total number of points that it has pruned, and the leader machine keeps track of ρ, the total number of points it received as potential outliers. Periodically the leader polls the workers for their values of τ_i's. Whenever $\sum_{i=1}^{n} \tau_i + \rho = |D|$, the leader sends a terminate message to all the machines. Alternatively, each machine can send a termination signal to the leader when the remaining test block size becomes zero.

A. Fault Isolation

In *FDCS*, it is fairly easy to isolate the attribute or feature which caused the outlier score to be high. Let x_t be the entity with the highest anomaly score (*i.e.* $\delta_k(x_t, D)$) and $y_1, y_2, \ldots, y_k$ be its k-nearest neighbors. Then, the anomaly score is:

$$\delta_k(x, D) = \frac{1}{k} \sum_{i=1}^{k} dist(x, y_i)$$

Algorithm 3: *FDCS* at master machine

Output: O_k, the set of outliers
Initialization: $O_k \leftarrow \emptyset$;
if *$(x, r_x, \mathcal{L}_k(x))$ is received* **then**
  $\rho \leftarrow \rho + 1$;
  if $|O_k| \leq t - 1$ **then**
    Add x to O_k;
  if $|O_k| = t - 1$ **then**
    $c \leftarrow \min\{\delta_k(y, D) : y \in O_k\}$;
    Broadcast c to all machines;
  if $|O_k| \geq t$ **then**
    if $r_x > \min\{\delta_k(y, D) : y \in O_k\}$ **then**
      Drop $y \in O_k$ with minimum δ_k;
      Add x to O_k;
      $c \leftarrow \min\{\delta_k(y, D) : y \in O_k\}$;
      Broadcast c to all machines;

where $dist(x, y_i)$ is the squared euclidean distance between x and y_i:

$$dist(x, y_i) = \sum_{j=1}^{d} \left(x^{(j)} - y_i^{(j)} \right)^2$$

This shows that the overall score can be decomposed amongst its individual components and the contribution of the j-th ($j = 1 : d$) variable towards the outlier score is:

$$\frac{1}{k} \sum_{i=1}^{k} \left(x^{(j)} - y_i^{(j)} \right)^2 .$$

This is the quantity that we have used in our experiments as the contribution of the j-th feature towards the overall score.

B. Efficient Preprocessing for Faster Computation

It has been shown earlier in [10][13] that distance based algorithms suffer from computational overhead due to its po-

tential quadratic time complexity. To overcome this, Bhaduri *et al.* [10] proposed a novel reordering technique of the data. In the main technique, the test points are ordered according to their distance to a fixed (randomly chosen) point in space, with the largest being the one tested first. Moreover, when searching the k-nn of a single point, the data is processed in a spiral fashion as shown in Fig. 2. They have shown that this search strategy exploits better spatial locality, and therefore, shorter running times. Also by ordering the test points in largest to smallest distance to a fixed point, it is intuitive that the cut off may increase faster, resulting in better pruning. We have used this index at each machine of our distributed algorithm to execute the local computation faster.

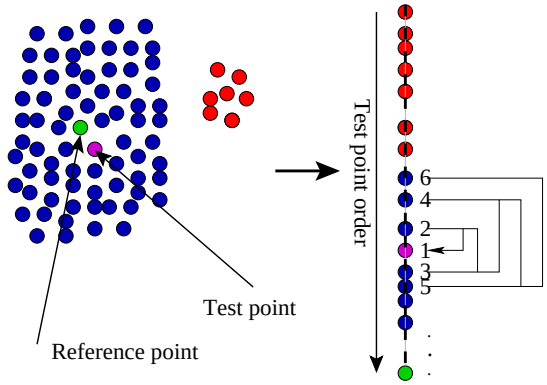


Figure 2. Description of the index. Left figure shows a dataset with normal points in blue, outliers in red and the reference and test point. The right figure shows the order in which the test points are processed with the points farthest from the reference point being processed first.

V. Experiments

In this section we describe an empirical evaluation of our *FDCS* algorithm.

A. Infrastructure Description

FDCS algorithm is implemented in C/C++ using MPI architecture for message passing. We have run all our experiments in a cluster infrastructure at NASA containing 128 nodes with 16 machines each having two, quad core Intel Xeon 2.66 GHz processors and 8 GB of memory, running Red Hat Linux. Cluster jobs are managed by the open source torque PBS scheduler. All machines have an NFS mounted raid array for data storage from a central machine connected through Gigabit Ethernet.

B. Data Description

The data collection for this experiment has been done using the cluster performance parameters recorded by the Ganglia monitoring system version 3.0.7. There are a total of 30 parameters measured here which cover different performance aspects of the cloud (cluster in this case) such as CPU usage, RAM usage, disk access, secondary memory access, job submission time, job completion time, boot time of the machine and so on. The parameter list is shown in Table V-C. The system monitors the performance parameters every 15 seconds and logs the average for each 6 minute interval. The files are exported daily at this resolution in comma separated format.

C. Experimental Setup

For our experiments, we monitored the cluster performance in a controlled environment by submitting a fixed set of 64 jobs to run on 8 machines for 3 days. Our job consists of reading 200 MB numerical data followed by a kernel and SVD computation, and finally writing the solution on disk files. The code written in MATLAB is shown in Figure 3. The *FDCS* algorithm in our experiment uses the last 27 parameters.

D. Results

The cluster performance data for each of the 8 machines concatenated as 6 minute composites for 3 days is stored locally at each machine and we run the distributed *FDCS* algorithm on this data to identify the top 50 global outliers. The outliers identified are unique <machine-id-time interval> tuples in this data set. The report generated by *FDCS* identifies the most frequent machine id in this list of top 50 outliers and returns that machine as the highest ranked faulty machine for the given job and time period. Figure 4 shows a possible report generated as the output of the *FDCS* algorithm. The report lists the top k (user specified) number of anomalies from the entire data set. For each of the anomalies, the algorithm computes the anomaly scores and also the respective weights associated with the parameters responsible for the anomalous behavior. The histogram on the right of Figure 4 shows the counts of the most anomalous machines in the top k list. The most frequently occurring machine id in the top k list is designated as the most faulty machine in the list.

The *FDCS* algorithm can not only identify the most faulty machine for a job, but also can isolate the cause of the fault by indicating the parameter which behaves in the most erratic fashion compared to the others. In our analysis, the cluster shows no signs of anomaly and, therefore, we have artificially injected faults for demonstration purposes. We have made the free swap space of machine number 8 decrease by 80% for 10 consecutive intervals towards the tail end of the job and then run the *FDCS* algorithm on this data set. We see that the algorithm reports machine 8 as the most anomalous machine and the free swap space and the processor load as the two most anomalous features in the data set. Figure 5 shows the plot of these two features for the entire job span. The red curve represents the time series for machine 8 while the blue curve represents the most normal time series for the same feature. We call the machine with

job schedule	date, time, boottime
network	bytes_in, bytes_out, pkts_in, pkts_out
processor	cpu_aidle, cpu_idle, cpu_nice, cpu_num, cpu_speed, cpu_system, cpu_user, cpu_wio
process	load_fifteen, load_five, load_one, proc_run, proc_total
main memory	mem_buffers, mem_cached, mem_free, mem_shared, mem_total, part_max_used
storage	disk_free, disk_total, swap_free, swap_total

Table I
LIST OF PERFORMANCE PARAMETERS OBTAINED USING GANGLIA

```
tic
SRCDir=inStruct.SRCDir;
filelist=dir([SRCDir,'*.mat']);
filelist={filelist.name};
K=zeros(length(filelist));
%% Read Files and build sub kernel %%
for i=1:length(filelist)
    Flight1=load([SRCDir,filelist{i}]);
    for j=1:length(filelist)
        Flight2=load([SRCDir,filelist{j}]);

K(i,j)=mean(mean(exp(Flight1.Flight.data(:,1:10))))+mean(mean(exp(Flight2.Flight.data(:,1:10))));
    end
end
Results.Runtime.ReadBuildSubKernel=toc;
%% Build full kernel%%
Kfull=zeros(length(filelist)*6);
count=1;
for i=1:6
     for j=1:6
         if(mod(count,2))

Kfull((i-1)*length(filelist)+1:i*length(filelist),(j-1)*length(filelist)+1:j*length(filelist))=K;
         else

Kfull((i-1)*length(filelist)+1:i*length(filelist),(j-1)*length(filelist)+1:j*length(filelist))=inv(K);
         end
     end
end
%% Solve for SVD%%
[U,S,V]=svd(Kfull);
Results.Runtime.BuildBigKSolveSVD=toc-Results.Runtime.ReadBuildSubKernel;
%% Write out SVD%%
csvwrite(['/data2/bmatthew/IDU_Cluster_Test/U',num2str(inStruct.pNum),'.csv'],U);
csvwrite(['/data2/bmatthew/IDU_Cluster_Test/S',num2str(inStruct.pNum),'.csv'],S);
csvwrite(['/data2/bmatthew/IDU_Cluster_Test/V',num2str(inStruct.pNum),'.csv'],S);
Results.Runtime.WriteOutResults=toc-Results.Runtime.BuildBigKSolveSVD;
```

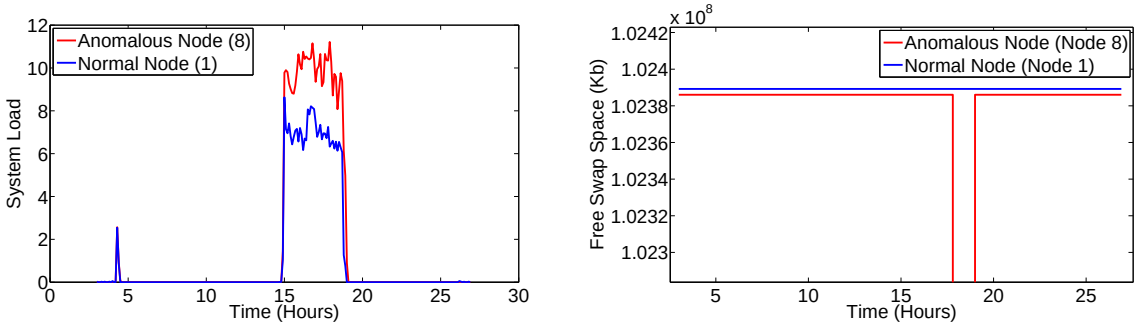
Figure 3. Matlab code for fictitious job used to measure cluster performance

the lowest frequency of occurrence in the top k list as the most normal machine.

In another scenario, we have run our experiment in a regular cluster environment with multiple other jobs running simultaneously. For this data, we identify the features that occur the maximum number of times in the top k anomaly list. The processor load and memory cache appear to be the two most frequent parameters identified to be most anomalous. Figures 6 and 7 shows the time series of both of these features for the entire 26 hour period that we have monitored the cluster system. This experiment validates that the anomalies identified by the centralized and the distributed algorithm are identical. Most of the anomalies for the cache memory are outside of our submitted job execution indicating that the other job(s) running must have been extremely memory intensive. On the other hand, quite a few anomalies in the processor load variable occur during the execution of our submitted job, indicating that the out job is a computation intensive job adding to the processor load.

VI. CONCLUSION

Given a cloud infrastructure with hundreds to thousands of machines, it is always a challenge for the system administrators to monitor the health of the machines. Monitoring programs such as Ganglia only allow the administrators to visualize the performance of all the machines using a web based GUI. As the scale of the system increases, it is imperative to develop automated methods to detect the faulty machines and isolate the causes before these faults have cascading effects on the entire system. By replacing



(a) CPU load vs. time

(b) Free swap space vs. time

Figure 5. Time series plots of two most anomalous features for the most faulty machine identified by the *FDCS* algorithm

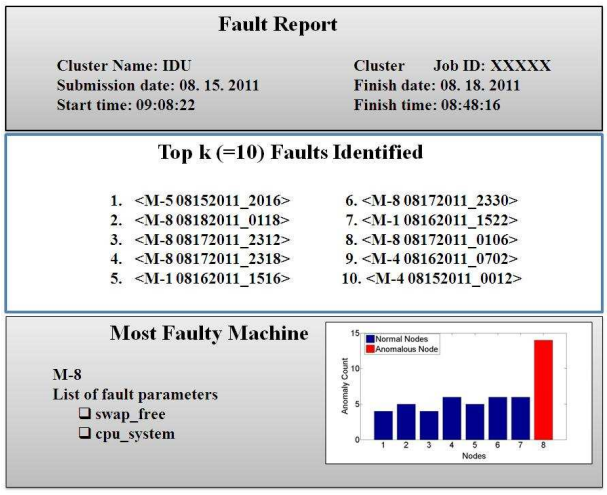


Figure 4. Sample report generated for identifying the top k outliers in the performance data. The report highlights the most faulty machine from the top k list.

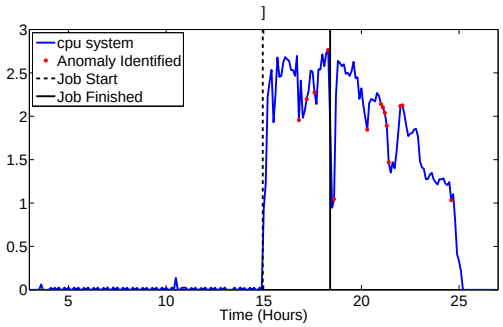


Figure 6. Time series plot of CPU load for the entire monitoring duration with anomaly time points highlighted for Machine 8

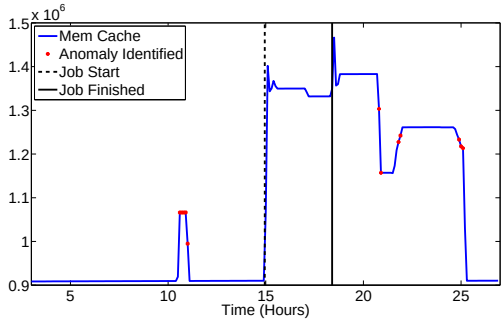


Figure 7. Time series plot of cache memory for the entire monitoring duration with anomaly time points highlighted for Machine 8

the human in the loop by an automated fault detection technique, the response time decreases dramatically. Our *FDCS* framework achieves this goal by deploying a distributed outlier detection algorithm that does not require data to be centralized, allowing extremely fast detection. *FDCS* has a reporting system which returns the top few faulty machines along with the reasons as to why they are faulty.

As part of future work, we plan to deploy this system to large production systems to test the performance of *FDCS*.

Acknowledgments

This research is supported by the NASA System Wide Safety Assurance Technologies project under NASA Aeronautics Mission Directorate.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," Electrical Engineering and Computer Sciences University of California at Berkeley, Tech. Rep. UCB/EECS-2009-28, 2009.

[2] J. Arshad, P. Townend, and J. Xu, "An Automatic Intrusion Diagnosis Approach for Clouds," *International Journal of Automation and Computing*, vol. 8, pp. 286–296, 2011.

[3] A. X. Zheng, J. Lloyd, and E. Brewer, "Failure Diagnosis Using Decision Trees," in *Proceedings of the First International Conference on Autonomic Computing*, 2004, pp. 36–43.

[4] H. Chen, G. Jiang, and K. Yoshihira, "Failure Detection in Large-Scale Internet Services by Principal Subspace Mapping," *IEEE Trans. on Knowl. and Data Eng.*, vol. 19, pp. 1308–1320, 2007.

[5] M. Isard, "Autopilot: Automatic Data Center Management," *Operating Systems Review*, vol. 41, pp. 60–67, 2007.

[6] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: automated classification of performance crises," in *Proceedings of EuroSys'10*, 2010, pp. 111–124.

[7] M. Goldszmidt, D. Woodard, and P. Bodik, "Real-time identification of performance problems in large distributed systems," in *Machine Learning and Knowledge Discovery for Engineering Systems Health Management*, A. Srivastava and J. Han, Eds. Taylor and Francis, 2011.

[8] D. Pelleg, M. Ben-Yehuda, R. Harper, L. Spainhower, and T. Adeshiyan, "Vigilant: out-of-band detection of failures in virtual machines," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 26–31, 2008.

[9] N. Palatin, A. Leizarowitz, A. Schuster, and R. Wolff, "Mining for misconfigured machines in grid systems," in *Proceedings of KDD'06*, 2006, pp. 687–692.

[10] K. Bhaduri, B. Matthews, and C. Giannella, "Algorithms for Speeding up Distance-Based Outlier Detection," in *Proceedings of KDD'11*, 2011, pp. 859–867.

[11] M. Gabel, R. Gilad-Bachrach, N. Bjorner, and A. Schuster, "Latent Fault Detection in Cloud Services," Microsoft Research, Tech. Rep. MSR-TR-2011-83, 2011.

[12] S. Bay and M. Schwabacher, "Mining distance-based outliers in near linear time with randomization and a simple pruning rule," in *Proceedings of SIGKDD'03*, 2003, pp. 29–38.

[13] M. Otey, A. Ghoting, and S. Parthasarathy, "Fast Distributed Outlier Detection in Mixed-Attribute Data Sets," *DMKD*, vol. 12, pp. 203–228, 2006.