Mining for Misconfigured Machines in Grid Systems

Noam Palatin, Arie Leizarowitz
Math Dept.
Technion – Israel
{noamp,la}@tx.technion.ac.il

Assaf Schuster, Ran Wolff
CS Dept.
Technion – Israel
{assaf,ranw}@cs.technion.ac.il

ABSTRACT

Grid systems are proving increasingly useful for managing the batch computing jobs of organizations. One well-known example is Intel, whose internally developed NetBatch system manages tens of thousands of machines. The size, heterogeneity, and complexity of grid systems make them very difficult, however, to configure. This often results in misconfigured machines, which may adversely affect the entire system.

We investigate a distributed data mining approach for detection of misconfigured machines. Our Grid Monitoring System (GMS) non-intrusively collects data from all sources (log files, system services, etc.) available throughout the grid system. It converts raw data to semantically meaningful data and stores this data on the machine it was obtained from, limiting incurred overhead and allowing scalability. Afterwards, when analysis is requested, a distributed outliers detection algorithm is employed to identify misconfigured machines. The algorithm itself is implemented as a recursive workflow of grid jobs. It is especially suited to grid systems, in which the machines might be unavailable most of the time and often fail altogether.

Categories & Subject Descriptors

C.2.4 Distributed Systems; H.2.8 Database Applications—Data mining

General Terms:

Algorithms, Management, Performance, Human Factors

Keywords:

Grid Systems, System Monitoring, Grid Information System, Distributed Data Mining, Outliers Detection

1. INTRODUCTION

Grid systems have developed, over the last fifteen years, as a natural extension to both high-performance cluster technology and batch management systems. Their supreme flexibility and scalability have proved crucial for the management of organizational computing power in today's data immersed and computationally intensive business arena. Grid systems manage pools of heterogeneous machines (e.g., servers and workstations) which can be harnessed for the execution of jobs submitted by any user in the organization. The number of machines controlled by a single system can sometimes reach tens of thousands — e.g., Condor [7], with up to 20,000 machines at U. Wisconsin and NetBatch, with as many as 35,000 machines at Intel. These systems often manage computers located in several geographically distant sites and organized in pools containing up to thousands of machines each.

Grid systems are notoriously difficult to configure. Every installation of a grid system is slightly different because the organization of each pool — the placement and number of services, for example — reflects the network topology at the site. Furthermore, the machines managed by a typical grid system can differ greatly. Finally, many of the resources available to the different machines are time varying (e.g., software licenses expire and storage space fills up over time). Hence, the system administrators must configure many attributes, and each site might suffer very different problems. The configuration task requires that administrators have a sound understanding of both the grid system's internals and their own site's organization. It is a complex and error-prone task.

Misconfigured machines, together with machines with faulty hardware or buggy software, can lead to so-called *'black holes'* (machines that accept many jobs and fail to complete any of them) or to other irregular machine behavior. At worst, a misconfigured machine can obstruct the work of the entire organization. A more ubiquitous effect of misconfigured machines is the reduction of system *goodput* — "the allocation time when a remotely executing application uses the CPU to make forward progress" [2].

To identify misconfigured machines, most organizations rely on user feedback and domain experts' manual analysis of log files. However, many problems pass unnoticed or unreported, rendering analysis nearly infeasible. Even when experts are aware of a problem, the information essential for detecting a misconfigured machine might be divided among different machines in the pool. Within each such machine, the details required for the analysis of the problem might be hidden in the vast number of configuration attributes or event logs of the grid system and the multiple services (NFS, DFS, JVM, etc.) that usually interact with a large system. All of this makes manual analysis time consuming, error-prone, and exhausting.

The dominant approach for automatizing misconfiguration detection in enterprise computing systems is a rule-based expert system. A typical system employs the knowledge of domain experts to construct a set of rules for identifying problems and taking ade-

KDD'06, August 20–23, 2006, Philadelphia, Pennsylvania, USA.

quate corrective actions. However, expert systems have some limitations: because the rules are constructed manually, their accuracy and completeness are limited. Furthermore, experts' time is scarce and very expensive, and the rules are very difficult to maintain (e.g., when the system moves to a new version).

In contrast to expert systems, data mining algorithms extract useful information from large data sets with limited, if any, prior knowledge about the data. Therefore, we propose that a tool using data mining techniques can outperform rule-based tools in misconfiguration detection. If most of the machines in a well maintained pool are properly configured, normative behavior can be modeled and misconfigured machines detected as outliers.

We implemented the Grid Monitoring System (GMS) – an extension to the Condor grid system which automatically detects oddly behaving machines. The challenges in building GMS relate to both system architecture and data mining. On the system side, the challenges are the immensity of the data and the limited system resources allocated for monitoring, along with the requirement that the Condor system not be changed — i.e., reliance on existent data sources. We address the former problem by building a fully distributed information system in which data is not moved from its origin — resulting in GMS overhead no greater than that required by existing logging facilities. To make use of existing data sources we implemented a mechanism that transforms the data from its raw form to an ontologically meaningful one.

On the algorithmic side, a distributed outlier detection algorithm was implemented in which every participant (i.e., computer in the grid system) can proceed at a different pace – based on its availability and the data it has. This allows outliers to be detected even if most of the normative machines are not available. The algorithm was implemented using recursive calls for Condor workflows. Analysis is initiated by submitting a Condor job, and it progresses through the automated submission of further jobs. To the best of our knowledge, our algorithm is the first distributed data mining algorithm that operates in a grid environment. Additionally, it is one of the first applications of data mining for solving a grid related problem.

We experimented with GMS in a pool of 42 heterogeneous machines, four of which were indicated by GMS to be misconfigured. The information provided by GMS aided in the investigation of the strange behavior resulting from 3 real misconfigured machines, and in determining the causes of the problem. Further experiments verified that GMS is scalable and suitable for operation in a real-life system.

The rest of this paper is organized as follows: The next section presents a brief review of related works in the areas of automated failure analysis and outlier detection, as well as a brief introduction to grid systems architecture. Section 3 describes the data, its acquisition, and its preprocessing. Section 4 describes the analysis of the data and the distributed outlier detection algorithm. Section 5 briefly describes system architecture and implementation. Section 6 outlines our experiments. Finally, conclusions are drawn in section 7.

2. PRELIMINARIES AND RELATED WORK

2.1 System Misconfiguration Detection

There are two general approaches to automated system management: white-box and black-box. The white-box approach relies on knowledge of the system and its behavior. A typical white-box management system such as Tivoli's TEC interprets system events according to a set of rules. These rules specify exceptional behavior patterns and appropriate responses to them.

In the black-box approach, problems are detected and diagnosed with limited, if any, knowledge about the system. The approach is used to try and learn which behaviors are abnormal. The black box approach is implemented today in several systems. For example, eBay (see Chen *et al.* [5]) experimented with a system that diagnoses failures by training a degraded decision tree on the record of successful and failed network transactions. The main disadvantage of eBay's system is that each transaction is described by just six features. This is mainly because the system centralizes the data and processes it on-the-fly.

2.2 Outlier Detection

Hodge and Austin [6] define an outlier point as "... one that appears to deviate markedly from other members of the sample in which it occurs." Of course, this definition leaves a lot to interpretation. This is not accidental, in the absence of a model for the data, the definition of an outlier is necessarily heuristic. Of the many proposed heuristics, this paper focuses on unsupervised methods, which are less demanding of the user. Specifically, we choose to focus on distance-based outlier detection methods, in which the main responsibility of the user is to define a distance metric on the domain of the data.

In GMS, we use the definition proposed by Angiulli et al. for the HilOut Algorithm [1]. By this definition the outlier score of a point is its average distance to its m nearest neighbors. A distributed version of the HilOut algorithm can be found in the literature [3], in the context of Wireless Sensor Networks (WSN). Grid systems, however, are different from WSNs in that the main challenge they pose is not reducing messaging but rather addressing the limited availability of resources. Thus, we describe a different distributed scheme for HilOut, which is innovative in its own right.

2.3 Grid Systems

Grid systems manage and run user jobs on a collection of computers called pools, which contain two main types of machines: *execution machines* and *submission machines*, and a handful of other machines which provide additional system services. Submission machines serve as proxy for the system users, who submit their jobs to the grid system through these machines, which then manage job execution. Execution machines share their computational capabilities with the pool according to the policies of the machine's owner. As both submission and execution machines pose requests and requirements, the task of matching between jobs and machines requires a third component — the matchmaker. The matchmaker collects information about all the pool participants, calculates the satisfiability of their demands, and notifies the submission and the execution machines of potentially compatible partners. One last grid service, of which GMS takes advantage, is the ability to plan and execute workflows – groups of jobs which depend on one another.

3. ACQUIRING, PREPROCESSING, AND STORING DATA

The quality of any data mining process is known to be critically dependent on data acquisition and preprocessing. Furthermore, the organization of the data is crucial in determining which algorithms can be used and, consequently, the performance of the process. This section outlines the main points of interest in the GMS approach to data acquisition, preprocessing, and organization.

3.1 Data sources and acquisition

There are two main approaches to system data acquisition: intrusive and non-intrusive. Intrusive data acquisition requires the

manipulation of the system for the extraction of data. The non-intrusive approach, which is the one employed by GMS, prefers using existing data sources. This is both because those sources include data about high level objects (jobs, etc.) and because the overhead incurred by monitoring is thus minimized. The sources used by GMS are:

- Log files: every component of the grid system is represented by a daemon. Daemons log their actions in dedicated log files, mainly for the purpose of debugging. This file also contains timestamps, utilization statistics, error messages, and other important information. Log files are read as event streams.
- Utilities: Grid systems supply utilities which extract useful information about the state of each machine, the overall status of the pool, machine configurations, and information about the jobs. Utilities are used through sampling — they are called periodically and can be matched against log file data, primarily by using time-stamps.
- Configuration files: Each grid system has configuration files which contain hundred of attributes. These files are read (once) and the data is stored in reference tables.

3.2 Preprocessing

Preprocessing of data in GMS mainly takes the form of converting raw data into semantically meaningful data. This is done by translating it into ontological terms. The use of ontologies in data mining has been discussed widely in the literature and has three main benefits. First, it increases the interpretability of the outcome. Second, it enriches the data with background information. Third, it allows data reduction by focusing on those parts of the data the ontology architect deems more important. Additionally, from a systems design perspective, the use of ontology allows the data to be virtualized, resulting in easy porting of the higher levels (the analytical part) of GMS from one grid system to the another.

The ontology used in GMS follows the principles suggested by Cannataro *et al*. [4]. It is hierarchical, with the highest level of the hierarchy containing the most general concepts. Each of these concepts is then broken down into more subtle concepts until, at the lowest level, *ground concepts* or *basic concepts* are considered. The relation between levels is that an upper level concept is defined by the assignment of its values to lower level concepts.

Besides translation into ontological terms, GMS preprocessing also addresses missing values. We treat missing values that are never available in some architectures as a value in their own right, and purge altogether records that contain missing values of temporarily unavailable concepts.

3.3 Data Organization

The data in GMS is fully distributed. The reasons for this are the desire to restrict to a minimum the GMS overhead while no analysis is taking place, and the need for scalability with the number of machines. Regarding the latter, it is enough to note that the average data rate at an execution machine can top one kilobyte per second. Accumulating this data over a 24 hour sliding window in a 1,000 machine pool would yield more than one hundred gigabytes of raw data, an amount requiring specialized resources for management if centralized. However, if the data remains distributed, the amount per execution machine — a mere hundred megabytes — can be supported by off-the-shelf, even free, databases such as MySQL.

4. DATA ANALYSIS

4.1 General approach

Two major assumptions guide our approach to detecting misconfigured machines: First, we assume that the majority of machines in a well-maintained pool are properly configured. Second, we assume that misconfigured machines behave differently from other, similar machines. The first of these assumptions limits our approach to systems that are generally operative, and predicts that it would fail if most of the resources are misconfigured. The second assumption limits the usefulness of GMS to misconfigurations that affect the performance of jobs (and not, e.g., system security).

Our choice of an algorithm is strongly influenced by two computational characteristics of grid systems. First, function shipping in grid systems is, by far, cheaper than data shipping — i.e., it pays to process the data where it resides rather than ship it elsewhere for processing. This, together with the difficulty in storing centralized data (as discussed in the previous section), motivates a distributed outlier detection algorithm. Second, machines in a grid system are expected to have very low availability. Thus, the algorithm has got to be able to proceed asynchronously and produce results based on the input of only some of the machines.

Finally, our approach is influenced by characteristics of the data itself: It takes many features to accurately describe events which occur in grid systems, and those events are very heterogeneous. This means that the data is extremely sparse and its distribution intricate. To overcome data sparsity, one can focus on jobs that are associated with a particular application getting varied input parameters. In our implementation, we emulate this kind of job by executing standard benchmarks with random arguments.

4.2 Notation

Let $P = \{P_1, P_2, \dots\}$ be a set of participants in the algorithm, and let $S_i = \left\{\vec{x_i^1}, \vec{x_i^2}, \dots\right\}$ be the input of participant P_i. Each input tuple $\vec{x_i^j}$ is taken from an arbitrary metric space $\mathbb{D}$, on which the metric $d : \mathbb{D} \times \mathbb{D} \to \mathbb{R}^+$ is defined. We denote S_N the union of the inputs of all participants. In rest of this paper we assume the distances between points in S_N are unique[1]. Among other things, this means that for each $S \subseteq S_N$ the solution of the HilOut outlier detection algorithm is uniquely defined.

For any arbitrary tuple $\vec{x}$ we define the *support* of $\vec{x}$, $[\vec{x}|S]_m$ to be the set of m points in S which are the closest to $\vec{x}$. For two sets of points $S, R \subseteq \mathbb{D}$ we define the support of R from S to be the union of the support from S for every point in R. We denote $\hat{d}(\vec{x}, S)$ the average distance of $\vec{x}$ from the points in S. Consequently, $\hat{d}\left(\vec{x}, [\vec{x}|S]_m\right)$ denotes the average distance of $\vec{x}$ from its m nearest neighbors in S. For any set S of tuples from $\mathbb{D}$, we define $\mathcal{A}_{k,m}(S)$ to be the top k outliers as computed by the (centralized) HilOut algorithm when executed on S. By definition of HilOut, these are k points from S such that for all $\vec{x} \in \mathcal{A}_{k,m}(S), \vec{y} \in S \setminus \mathcal{A}_{k,m}(S)$ we have $\hat{d}\left(\vec{x}, [\vec{x}|S]_m\right) > \hat{d}\left(\vec{y}, [\vec{y}|S]_m\right)$.

4.3 Algorithm

The basic idea of the Distributed-HilOut algorithm is to have the participants construct together a set of input points S_G from which the solution can be deduced. S_G has three important qualities: First, it is eventually shared by all of the participants. Second, the solution of HilOut, when calculated from S_G, is the same one which is calculated from S_N — $\mathcal{A}_{k,m}(S_G) = \mathcal{A}_{k,m}(S_N)$. Third,

[1]This assumption is easily enforced by adding a little randomness to the numeral features of each data point.

the support of the solution on S_G from S_G — $[\mathcal{A}_{k,m}(S_G)|S_G]_m$ — is the same as the support from the entire set of inputs S_N — $[\mathcal{A}_{k,m}(S_G)|S_N]_m$.

Since many of the participants are rarely available, the progression of S_G over time may be slow. Every time a participant P_i becomes available (i.e., a grid resource can accept a job related to the analysis), it will receive the latest updates to S_G and will have a chance to contribute to S_G from S_i. By tracking the contributions of participants to S_G, an external observer can compute an ad hoc solution to HilOut at any given time. Besides providing for temporal (sometimes lasting) non-availability of resources, the algorithm has two additional benefits: One, the size of S_G is often very small with respect to S_N, and the number of participants contributing to S_G very small with respect to the overall number of participants. Two, $\mathcal{A}_{k,m}(S_G)$ often converges quite quickly, and the rest of the computation deals solely with the convergence of $[\mathcal{A}_{k,m}(S_G)|S_G]_m$. $\mathcal{A}_{k,m}(S_G)$ converges quickly because many of the well-configured machines could be used to point out a specific outlier.

The details of the Distributed HilOut algorithm are given in Algorithms 1 through 3. The algorithm is executed by a sequence of recursive workflows. The first algorithm, Algorithm 1, is run by the user. It submits a workflow (Algorithm 2) to every resource in the pool and terminates. Afterwards, each of these workflows submits a job (Algorithm 3) to its designated resource and awaits the job's successful termination. If the job returns with an empty output, the workflow terminates. Otherwise, it adds the output to S_G and recursively submits another workflow — similar to itself — to each resource in the pool.

Algorithm 1 Distributed HilOut – User Side

Input: The number of desired outliers – k, and the number of nearest neighbors to be considered for each outlier – m
Initialization:
Set $S_G \leftarrow \emptyset$
For every P_i submit a Distributed-HilOut workflow with arguments P_i, k, and m
On request for output: Provide $\mathcal{A}_{k,m}(S_G)$ as the ad hoc output.

Algorithm 2 Distributed-HilOut Workflow

Arguments: P_i, k, and m
Submit a Distributed-HilOut Job to P_i with arguments k, m, and S_G
Wait for the job to return successfully with output R
Set $S_G \leftarrow S_G \cup R$
If $R \neq \emptyset$ submit a Distributed-HilOut workflow for every $P_j \neq P_i$ with arguments P_j, k, and m

If there are points from S_i which should be added to S_G, they are removed from S_i and returned as the output of the job. This happens on one of two conditions: 1. When there are points in the solution of HilOut over $S_i \cup S_G$ which come from S_i and not S_G. 2. When there are points in the support from $S_i \cup S_G$ to the solution as calculated over S_G alone, which are part of S_i and not S_G.

Moving points from S_i to S_G may change the outcome of HilOut on S_G. Thus, the second condition needs to be repeatedly evaluated by P_i until no more points are moved from S_i to S_G. Strictly for the sake of efficiency, this repeated evaluation is encapsulated in a while loop; otherwise, the same repeated evaluation would result from the recursive call to Alg. 3.

Algorithm 3 Distributed HilOut Job

Job parameters: k, m, S_G
Input at P_i**:** S_i
Job code:
Set $Q \leftarrow \mathcal{A}_{k,m}(S_G \cup S_i)$
Do
– $Q \leftarrow Q \cup [\mathcal{A}_{k,m}(S_G \cup Q)|S_G \cup S_i]_m$
While Q changes
Set $R \leftarrow Q \setminus S_G$
Set $S_i \leftarrow S_i \setminus R$
Return with R as output

5. THE GMS SYSTEM

Architecturally, GMS is divided into two parts. One part – the data collector – takes charge of data acquisition while the other part – the data miner – is in charge of analysis. The data collector is a stand-alone software component installed on the resources that are to be monitored by GMS. The data miner, on the other hand, is a grid workflow which executes jobs, exchanges data between resources, and produces the outcome.

The data collector siphons data from many of the data sources available on its machine and organizes the collected data into relational tables according to the ontology scheme. The data miner is implemented using Condor DAGs. The Condor DAGman is a workflow management engine. It supports execution of mutually dependent Condor jobs and limited control structures. In our implementation, there are two dependencies: The system submits a job to a specified execution machine and then waits for it to terminate successfully. Then, if the job returns non-empty output, the workflow will dictate that a number of new workflows be instantiated – one for every execution machine. These new workflows, in turn, repeat the same process recursively. It should be noted that the use of DAGs means that Condor itself takes charge of the server side of the data mining algorithm.

6. EVALUATION

To validate the usefulness of GMS we conducted an experiment in a pool of 42 heterogeneous Linux machines (84 virtual machines in all): 10 dual Intel XEON 1800 MHz machines with 1GB RAM, 6 dual Intel XEON 2400 MHz with 2GB RAM, and 26 dual IBM PowerPC 2200 MHz 64-bit machines with 4GB RAM.

We ran two benchmarks independently: BYTEmark – a benchmark that tests CPU, cache, memory, integer and floating-point performance – and Bonnie, which focuses on I/O throughput. We sent multiple instances of these benchmarks as Condor jobs to every machine in the pool, varying their arguments randomly across a large range. In all, an order of 9,000 jobs were executed. We then independently ran Distributed HilOut on the two resulting datasets. After preprocessing the dataset, we selected 56 attributes, which describe job execution and the properties of the execution machine. Job execution attributes include, for example, runtime, while the machine properties include attributes such as CPU architecture, memory size, disk space, and Condor version.

We used the following distance metric d: First, all of the numerical features were linearly normalized to the range $[0, 1]$, so that the weight of different attributes would not be influenced by range differences. Given two points, the distance between numerical features was calculated using a weighted L2 norm – with more weight allocated to job runtime indicators and to configuration features (memory size, etc.). This different weighting encourages the algorithm to gather records of jobs which ran on similar machines

Rank	Machine	Score	Four Main Contributing Attributes
1	bh10	0.476	SWAPIN, SWAPOUT, AVGCPU_SYS, AVGCPU_USER
2	bh10	0.431	SWAPOUT, SWAPIN, AVGCPU_USER, MAXCPU_USER
3	i4	0.425	DISKUSEDROOT, MAXCPU_USER, AVGCPU_USER, BUFFEREDMEM
4	i4	0.422	DISKUSEDROOT, MAXCPU_USER, AVGCPU_USER, AVGCPU_IDLE
5	bh13	0.422	LAUNCHERLEN, CPUMODEL, JOBLEN, AVGCPU_USER
6-8	i4	...	...
9	i3	0.421	AVGCPU_USER, MAXCPU_USER, MAXROOTUSED%, AVGROOTUSED%

Table 1: The output of Distributed HilOut on BYTEmark data.

Number of machines	$\|S_N\|$	$\|S_G\|$	Percent
10	751	190	25%
20	1470	336	23%
30	2290	428	19%

Table 2: Scalability with the number of machines and data points – BYTEmark benchmark. $k = 7$.

Number of machines	$\|S_N\|$	$\|S_G\|$	Percent
10	828	186	22%
20	1677	373	22%
30	2343	452	19%

Table 3: Scalability with the number of machines and data points – Bonnie benchmark. $k = 7$.

k	$\|S_N\|$	$\|S_G\|$	Percent
3	2290	304	13%
5	2290	369	16%
7	2290	428	19%

Table 4: Scalability with the number of outliers (k) – BYTEmark benchmark, 30 machines.

k	$\|S_N\|$	$\|S_G\|$	Percent
3	2343	287	12%
5	2343	372	16%
7	2343	452	19%

Table 5: Scalability with the number of outliers (k) – Bonnie benchmark, 30 machines.

and have similar runtime. Nominal features contributed zero to the overall distance if their value was the same, and a constant otherwise. This is with the exception of the machine identifier. This field contributed a very large constant if it was the same and zero otherwise. The reason is that we wanted neighbor points to belong to different machines – so that our algorithm would detect exceptional machines rather than exceptional executions in the same machine.

Below, we describe three different experiments. We note that between experiments some of the misconfigured machines were fixed, which explains the differences in the number of outlying machines, data points, etc. Although experimentally undesirable, this is an unavoidable outcome of working in a truly operational system. Finally, we note that in all our experiments the number of neighbors (m) was set to five.

6.1 Qualitative Results

We ran Distributed HilOut separately on the records of each benchmark. The outcome of the analysis was a list of suspect machines. Additionally, as shown in Table 1, the algorithm ranked for each outlier the main attributes that contributed to the high score. In both tests two machines, i3 and bh10, were indicated as misconfigured. Additionally, the test based on BYTEmark data indicated that bh13 and i4 were also outlying. With the help of a system administrator, we analyzed the four machines that had the highest ranking.

The machine bh10 was ranked highest because of excessive swap activity, but we were not able to recreate the phenomenon and so concluded that it was temporary.

The next highest ranked machine was i4, which contributed five of the top nine outlying points. In all of the outlying executions, the extraordinarily low user CPU was one of the outstanding attributes. A quick check found that the CPU load of that machine was very high. The source of the high load turned out to be a network daemon (Infiniband manager) that was accidentally installed on the machine. As a result, the user was only allocated a small percentage of the CPU time. After the system administrator shut the daemon down, the machine started to behave normally.

The third ranked machine was bh13. With that machine the algorithm indicated a mismatch of the CPU model and the time it takes to launch a job. As it turned out, this machine had a wrong BIOS setup: It was configured with active HyperThreading, which meant each CPU (the machine had two) was represented as two CPUs with half the system resources (memory, etc.). Consequently, it launched jobs slower than other machines with same CPU model.

The fourth ranked machine was i3. Here, GMS indicated a higher than usual use of the root file system. We found the root file system was nearly full. This led to the failure of the benchmark, and thus to much shorter runtime than usual.

Although qualitative, we consider the validation process highly successful. Of the four highest ranked machines, three were found to actually have been misconfigured. GMS contributed to the analysis of all three by pointing out not only which machines to check but also which attributes in the outlying machine differ from those in comparable machines. Further analysis of the misconfiguration, beyond using GMS, required no access of logs or configuration files. On the flip side, the Bonnie benchmark missed two out of the three misconfigured machines. We attribute this to the narrow nature of this benchmark, which focuses on I/O.

6.2 Quantitative Results

The goal of our second experiment was to evaluate the scalability of the Distributed-HilOut algorithm. Specifically, we wanted to examine what portion of the entire data set, S_N, is collected into S_G. To be scalable, S_G needs to grow sublinearly with the number of execution machines and at most linearly with the number of desired outliers – k.

Tables 2 and 3 depict the percentage of points — out of the total points produced by 10, 20, or 30 machines — collected into S_G.

As the number of machines grows, that percentage declines. In a large-scale setup with hundreds of execution machines, we expect the percentage to decline much further.

Figures 4 and 5 depict the percentage of points — out of the total points produced by 30 machines — that were collected when the user chose to search for 3, 5, or 7 outliers. That percentage increases linearly. We conclude that our approach is indeed scalable.

6.3 Interoperability

Our final set of experiments validated the ability of GMS to operate in a real-life grid system. We ran the Distributed HilOut algorithm with twenty of the well-configured machines shut down. Then, when no more workflows were pending for the active machines, we turned the rest of the machines on. The purpose of this experiment was to observe GMS behavior in the presence of failure. We noted the progression of the algorithm in terms of the recall (portion of the outcome correctly computed) and observed both the recall in terms of outlier machines (Recall – M) and in terms of data points (Recall – P). The data points are important because, for the same outlier machines, several indications of its misbehavior might exist, such that the attributes explaining the problem differ from one point to another.

With respect to the recall, our expectations — fast convergence regardless of the missing machines – were met. As tables 6 and 7 show, the recall progressed quickly. Furthermore, nearly complete recall of the patterns was achieved in both benchmarks, and all of the outlier machines were discovered. This means the administrators were able to start analyzing misconfigured machines almost immediately, even if many of the machines were unavailable. The computational overhead was not very large even after we turned the missing machine on again (as described in table 6 and 7, below the double line) — in all, about forty workflows resulted in additions to S_G.

7. CONCLUSION AND FUTURE WORK

We study the problem of detecting misconfigured machines in large grid systems. Because these systems are heterogeneous, a rich description of their operation is required for more accurate analysis. Moreover, their scale makes centralization of the data as well as its manual analysis inefficient. We therefore suggest a distributed architecture which enables automatic analysis of the data via data mining algorithms.

We implement a highly portable Grid Monitoring System (GMS) that relies on an ontology for virtualization of the underlying batch system and for enhancing data quality. We deploy our system on a heterogeneous Condor pool and demonstrate its effectiveness by discovering three misconfigured machines.

Outlier detection is just one of the algorithms which can be implemented on top of GMS and misconfiguration detection is just one of the possible applications of data mining for grid systems. We intend to extend GMS for various applications.

Time (h:mm)	$\lvert S_G \rvert$ / $\lvert S_N \rvert$	Workflows	Recall – P	Recall – M
0:00	0 / 4334	0	0 / 7	0 / 2
0:03	82 / 4334	6	4 / 7	1 / 2
0:07	460 / 4334	20	5 / 7	2 / 2
0:10	462 / 4334	22	6 / 7	2 / 2
0:23	495 / 4334	26	6 / 7	2 / 2
1:38	617 / 4334	37	7 / 7	2 / 2
2:54	619 / 4334	39	7 / 7	2 / 2

Table 6: Interoperability – a grid pool with twenty of the machines disabled – BYTEmark benchmark.

Time(h:mm)	$\lvert S_G \rvert$ / $\lvert S_N \rvert$	Workflows	Recall – P	Recall – M
0:00	0 / 4560	0	0 / 7	0 / 3
0:05	112 / 4560	4	2 / 7	1 / 3
0:07	417 / 4560	16	4 / 7	2 / 3
0:10	475 / 4560	22	5 / 7	3 / 3
0:15	478 / 4560	24	7 / 7	3 / 3
0:17	484 / 4560	27	6 / 7	3 / 3
0:31	494 / 4560	30	7 / 7	3 / 3
2:03	578 / 4560	40	7 / 7	3 / 3

Table 7: Interoperability – a grid pool with twenty of the computers disabled – Bonnie benchmark.

8. REFERENCES

[1] F. Angiulli and C. Pizzuti. Fast outlier detection in high dimensional spaces. In *Proc. of PKDD*, 2002.

[2] J. Basney and M. Livny. Improving goodput by co-scheduling CPU and network capacity. *Intl. Journal of High Performance Computing Applications*, 13(3), 1999.

[3] J. W. Branch, B. Szymanski, C. Giannella, R. Wolff, and H. Kargupta. In-network outlier detection in wireless sensor networks. In *Proc. of ICDCS*, July 2006.

[4] M. Cannataro, A. Massara, and P. Veltri. The OnBrowser ontology manager: Managing ontologies on the grid. In *Intl. Workshop on Semantic Intelligent Middleware for the Web and the Grid*, 2004.

[5] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *Proc. of ICAC*, 2004.

[6] Hodge V. and Austin J. A Survey of Outlier Detection Methodologies. *Artificial Intelligence Review*, 22:85–126, 2004.

[7] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A hunter of idle workstations. In *Proc. of ICDCS*, June 1988.

9. ACKNOWLEDGEMENT

This work was supported in part by the DataMiningGrid project – `www.DataMiningGrid.org`