# MINING SUPERCOMPUTER LOGS TO IDENTIFY EVENT CORRELATION

ADITYA PAKKI* AND HARSHITHA PARNANDI†

**Abstract.** In this project, we attempt to perform set mining, using two different techniques - clustering using DBScan algorithm and Apriori algorithm, on the logs written by Supercomputers. We compare the effectiveness of the rules generated as well as the time taken to run on the dataset of size over 600MB after running our pre-processing routines. Apart from the end goal of generating event correlations, we also performed studies on understanding the stability of DBScan algorithm as well as comparing it with the clustering techniques taught in class. In a very restrictive set of conditions described in this paper, we see DBScan clustering technique performing better than the Apriori algorithm in terms of generating relevant rules.

**Key words.** Analysis, event correlation, DBScan, Apriori, Intrepid, system logs.

**1. Problem Statement.** Efficiency as per Wikipedia, is defined as minimizing wastage of resources such as materials, energy, efforts, money or time in producing a desired result. The current fastest supercomputer on the planet, Tianhe-2, can execute over $33.86 * 10^{15}$ floating point operations per second (PFLOPS) against its peak capacity of 54.902 PFLOPS. The challenge is to keep these machines running for as long as possible for an amortized higher efficiency. In the push towards building a machine capable of executing one EFLOP per second (Exascale Computing), the way to higher efficiency, as predicted by High Performance Computing (HPC) researchers, is by minimizing the computation loss and down time due to system crashes. The observed Mean Time Between Failure(MTBF), a metric to measure the frequency of system crashes, of petascale machines is between 24 to 48 hours. However, the projected MTBF rates for an exascale machine is around 1 hour. Techniques other than checkpoint and restart are actively considered to improve the resiliency of the simulations. This field is an area of interest of one of the authors.

In this project, we used data mining techniques to study the feasibility of solving a part of the above problem. In our bigger picture, the goal is to build a failure predictor that triggers a preventive routine before a machine goes down. To perform such an event prediction, the errors have to be first categorized and then correlated to use as input for the prediction routine. We limit the scope of the project to event correlation for the course CS 6140 as can be seen in Figure 1. While each machine has a different architecture and different logging format, we worked on the system logs of Intrepid supercomputer [6]. The *key idea* we intend to learn from this project is to compare the effectiveness of two methods in identifying two events $E_z$ and $E_y$ correlated with confidence greater than 50%. While the confidence number might seem arbitrary, we hope that such a number might help us in finding another event $E_x$ which precedes events ($E_y$, $E_z$).

The rest of the paper is organized as follows - Section 2 describes how we processed the data,

[1]u0922291, pakkiadi@cs.utah.edu, School of Computing, University of Utah
[2]u0927529, pvharsh@cs.utah.edu, School of Computing, University of Utah

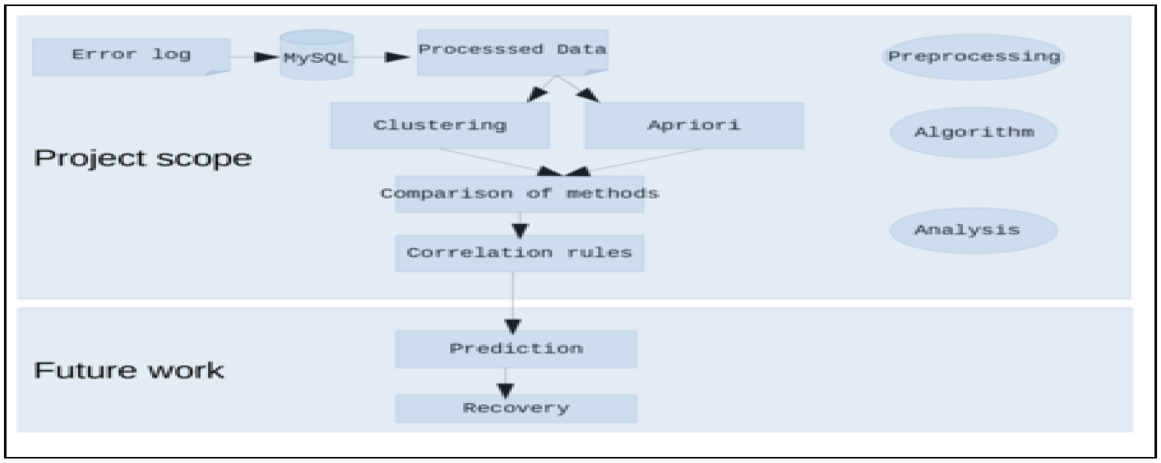
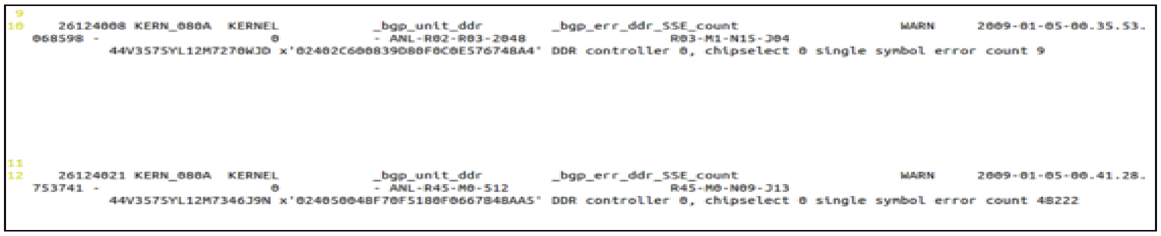


FIG. 1. *Project Scope and Future work*

Section 3 talks about our data mining algorithms - DBScan clustering and Apriori algorithm. In Section 4 has details about the project implementation and comparison of two techniques is in Section 5. The related work is described in Section 6 and we conclude with future work in Section 7.

**2. Data Processing.** Intrepid supercomputer [6] was Argonne Leadership Computing Facility in operation from 2008 to 2013 and was ranked the fourth fastest machine after it began operations. It has over 160,000 processors and had a peak performance of 458 TFlops. While the machine might be obsolete, analyzing the logs generated by it can give us insights on the volume and type of messages generated by such machines. The data was collected from two sources in the time period ranging from 05/01/2009 to 08/31/2009, the logs of the jobs from [3] having the details of all successfully completed jobs amounting to over 60,00 records. The source of the error logs is [1], that has over two million error messages. The Figure 2 shows two error messages in free text format and each message is considered an event, used interchangeably as error going forward. Events can be further classified into either trivial or warnings based on the SEVERITY field. Trivial messages can be events that poll the system's health, system boot up or periodic pings.


```
    26124008 KERN_080A  KERNEL           _bgp_unit_ddr        _bgp_err_ddr_SSE_count                 WARN     2009-01-05-00.35.53.
 068598 -                    0           - ANL-R02-R03-2048                         R03-M1-N15-J04
         44V3575YL12M7270WJD x'02402C600839D80F0C0E576748A4' DDR controller 0, chipselect 0 single symbol error count 9




    26124021 KERN_080A  KERNEL           _bgp_unit_ddr        _bgp_err_ddr_SSE_count                 WARN     2009-01-05-00.41.28.
 753741 -                    0           - ANL-R45-M0-512                           R45-M0-N09-J13
         44V3575YL12M7346J9N x'02405004BF70F5180F0667848AA5' DDR controller 0, chipselect 0 single symbol error count 48222
```

FIG. 2. *Sample error records in plain text format*

| Subcomponent | count | Subcomponent | count | Subcomponent | count | Subcomponent | count |
|---|---|---|---|---|---|---|---|
| bg_subcomp_e10k | 34 | bg_subcomp_linux | 3790 | BPC_CHIP | 20726 | BPC_LBIST | 46 |
| BPL_CHIP | 42 | BPL_LBIST | 252 | BRINGUP | 40 | COOKIE | 174 |
| DGEMM | 3241 | DMA | 24 | globalint_tst | 202 | HW_MONITOR | 39501 |
| InitNcs | 16 | MMCS_OPS | 906 | nodecard_env | 69 | PALOMINO | 991 |
| PALOMINO_L | 613 | PALOMINO_N | 4539 | PALOMINO_S | 113 | ReadLcInfo | 24 |
| UpdateNcInfoData | 553 | _bgp_unit_ciod | 902 | _bgp_unit_cns | 123 | _bgp_unit_collective | 59 |
| _bgp_unit_ddr | 59348 | _bgp_unit_dma | 390 | _bgp_unit_kernel | 1408 | _bgp_unit_serdes | 17 |

TABLE 2
*Fatal event subcomponents and counts*

The initial step to cleaning the data is achieved by loading both the logs into tables of MySQL database. We use the power of relational queries to narrow down the important dimensions of the error file. The following unique fields have been extracted - { Recid, Component, Node, Subcomponent, Severity, Event_time, Processor, Location, Message }. Given the wide range of errors that may be possible, we identify the relevant error severity types that can cause system failures as 'FATAL' that can cause system crashes. The counts are given in table 1. For these relevant error types, we query their frequent subcomponent types given by table 2.

| Severity | ERROR | **FATAL** | INFO | WARN |
|---|---|---|---|---|
| Count | 104802 | 33370 | 807651 | 419470 |

TABLE 1
*Error counts by severity*

The next step in defining the scope is to modify the error message fields to group similar events. The error messages we saw in the logs differ by constants. These error messages can be classifed as similar events if we replace the number with a fixed keyword or symbol. We first went through all the messages and then replaced all the numbers with 'd+' and the alphanumerics with the 'V*'.

Examples of messages templated to a fixed pattern.

- DDR controller 0, chipselect 0 single symbol error count
- DDR controller 1, chipselect 1 single symbol error count

will be replaced by "DDR controller **d+**, chipselect **d+** single symbol error count"

- BPC pin JK126, transfer 1, bit 106, BPC module pin V03, compute trace MEMORY1 DATA 79, DRAM chip U13, DRAM pin B9.

will be replaced by "BPC pin **V***, transfer **d+**, bit **d+**, BPC module pin **V***, compute trace DATA **d+**, DRAM chip **V***, DRAM pin **V***".

To say an event $E_z$ of severity type 'error' or 'fatal' is correlated with a event $E_y$, we first have to define a window of reference. The events within the window are considered relevant and occuring prior to the event $E_z$. For an event to be corelated, we first determine, all the events occuring with in the time window and store the time difference (in sec) between each unique event in a set. From the Figure 3 we see that, a majority of successful jobs have completed withtin 20 minutes of starting execution. Having a larger window, correlates unrelated events, and a smaller window might miss detecting possible event corelations. As window was a holistic measure, we came up with the value as two days and events within this time window preceding event $E_z$ are checked for correlation.

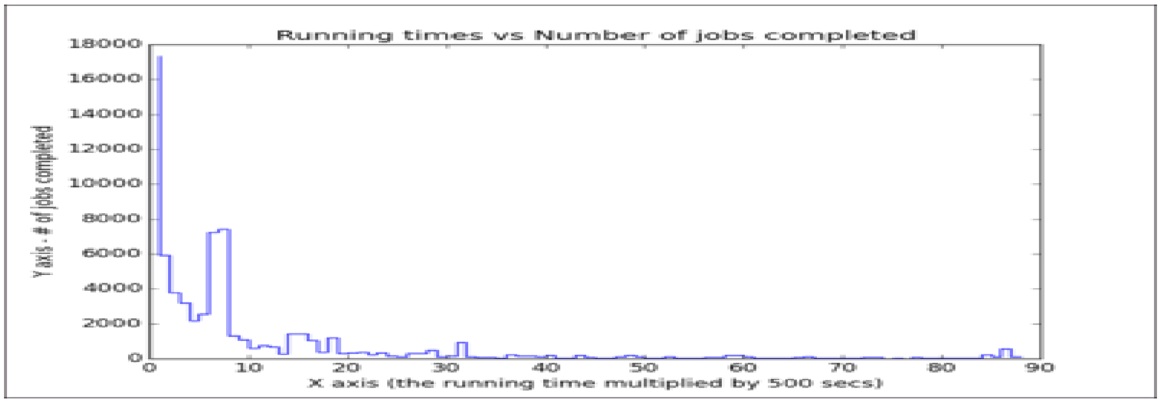


FIG. 3. *Running times of successful jobs above a threshold*

**3. Algorithm Motivation.** In this section, we breifly provide arguments for choosing the two algorithms for comparison over other available data mining techniques taught in the class [2]. Below are few of the properties of the data:

1. Insufficient knowledge on the approximate number of clusters. We limit the errors to severity type 'fatal'. From table 1, the upper bound could be over 137,000 clusters and if clustered based on subcomponents, we might get 28 clusters.
2. There is no notion of similarity in normal clustering techniques and can combine many interesting patterns when going higher up the chain in hierarchial clustering.
3. Dealing efficiently with noisy data is a property of DBScan [5] and not many clustering algorithms can deal with them.
4. Using any of the k-means and their medioid variety requires identifying the cluster centres. Running multiple times resolves the issue but can increase the computation cost of the algorithm.
5. Every data point in the dataset is ordered temporally and preserving this metric was considered important for our analysis.
6. One goal of our project has been to identify the frequent individual items and extend the chain as large as possible. Apriori Algorithm [4] fits this criteria perfectly.

After a brief literature survey, we felt DBScan clustering algorithm and Apriori algorithm to be a better fit to our problem. Below is the pseudo code for our correlation implementation.

### 3.1. DBScan Clustering Algorithm.

function **DBSCanCorr()** :
Input: list of messages M[], window W, minPts
Output: set of events S, delay set D

```
for all events preceding z in W:
  for all events y != z in W:
    find delays = (time(z) − timei(Y)) and c[] = delay counts
    if c[delay] > minPts:
      S.insert(A,B)
      D.insert(delay −start/2, delay + start/2)
return S, D
```

1. Adjust the $W$ and $minPts$ parameters for DBScan to generate meaningful correlations
2. Extract the top k correlated events for each event of type FATAL.

**3.2. Apriori Algorithm.**

function **AprioriCorr()** :
Input: list of messages M[][], window W, threhold T , subcomponent c
Output: set of events S

```
for each event of type c in Log:
  for all events preceding z in W:
    for k = 1, k < max(E_k size); k++:
      construct k−ary of unique events in W:
      if count of k subset(E_1 .. E_k) < T:
        discard rule
      else:
        S += {k subset of (E_1 .. E_k)}
return S
```

**4. Implementation Details.** In this section, we describe the implementation and experiments we tried for the project, with respect to the techniques taught in the class. Many details will be explained at the level mentioned in the class. After preprocessing the data and running queries on the table, we have limited the scope to identifying the correlation of events of severity type FATAL.

**Event de-duplication** The goal is to remove the periodic and repeated events. For repeated events, we first have to decide on the time within which events within the time is considered a repeat and discarded. We see the results as in table 3 that, the majority of the repeats are Info and Warn types and can be replaced with a single instance and there is not a huge difference in events repeating within 30 seconds or 60 seconds. There is no loss of information as the DBScan checks for previous occurences of record and we are storing the maximum difference in our dataset. Apriori also has no information loss as we are storing the unique record for every event and the set operation

| 60 secs | | | 30 secs | | |
|---|---|---|---|---|---|
| SNo | Count | Severity Type | SNo | Count | Severity Type |
| 1 | 15378 | INFO | 1 | 15378 | INFO |
| 2 | 15372 | INFO | 2 | 15372 | INFO |
| 3 | 15267 | INFO | 3 | 14241 | INFO |
| 4 | 14850 | WARN | 4 | 14119 | WARN |
| 5 | 10923 | INFO | 5 | 10920 | INFO |

TABLE 3
*Top 5 error that repeat - counts and severity*

| **Messages** | **GID** |
|---|---|
| Node d+ exited for reason d+ and code d+ and there was error d+ sending a message to the control V* | 26 |
| There were d+ proxy processes that did not cleanly at the end of job V* | 27 |
| Collective network receiver link d+ has spent d+ cycles resynchronizing with the sender | 28 |
| DDR controller single symbol error V* Controller d+ chipselect d+ V* | 29 |
| DDR controller chipkill error V* Controller d+ chipselect d+ V* | 30 |
| Error d+ reading message d+ from control V* | 31 |
| V* network link has been V* | 32 |
| Spurious Ethernet device interrupt V* | 33 |

TABLE 4
*Generating GID from error messages*

performs the same functionality.

**Event aggregation** The next step we perform is to replace the messages with a Group ID(GID). All messages having similar GID are considered similar. The goal in this step is minimizing the number of GID to classify the messages to fewer clusters. One straight forward approach is grouping all the items with similar subcomponent type. However, to understand event correlation, we were more concerned about the specific preceding event that triggers the event we are interested about. To this effect, we started replacing the constants in message fields to a templated character as seen in Section 2. The Jaccard similarity defined as $JS(A, B) = {}^{A \cap B}/_{A \cup B}$. Using a jaccard threshold $J_{th}$ of 0.75, we came up with 215 GID for varying severity types. The table 4 shows a couple of messages with their GIDs. Without using Jaccard similarity we get the following types of events, which could have been merged into two distinct events. Events 1 and 3 are similar and events 2 and 4 are a different event type.

1. V* instruction cache parity error has occurred
2. DDR controller single symbol error V* Controller d+ chipselect d+ V*
3. V* instruction cache parity error has occurred in TAG bit d+ V* V*
4. DDR controller double symbol error V* Controller d+ chipselect d+ V*

**Root cause Analysis** For the root cause analysis, we run the two algorithms, written in Python with database connectivity to MySQL database. The criteria on which we run our algorithms include

- subcomponent of type $\_bgp\_unit\_snoop$, $LINKCARD\_ENV$, InitNcs, $\_bgp\_unit\_serdes$ or ReadLcInfo, severity of event is FATAL and the window of reference is two days. For each such event, we run the DBScan algorithm separately and extract the events correlated with varying *minPts* sizes. We use confidence metric of 0.5, given as $P(A|B) = {}^{P(A \cap B)}/_{P(B)}$ to check for relevance where A and B are the number of times event A and event B occur. Comparison results can be seen in the next section.

## 5. Comparison Results.

## 6. Related Work.

## 7. Future Work and Conclusions.

## REFERENCES

[1] The computer failure data repository (cfdr). https://www.usenix.org/cfdr.

[2] Jeff philips course notes from course cs 6140 spring 2016. http://www.cs.utah.edu/ jeffp/teaching/cs5140.html.

[3] Logs of real parallel workloads from production systems. http://www.cs.huji.ac.il/labs/parallel/workload/logs.html.

[4] Wikipedia article about apriori algorithm. https://en.wikipedia.org/wiki/Apriori.

[5] Wikipedia article about dbscan clustering algorithm. https://en.wikipedia.org/wiki/DBSCAN.

[6] S. Kumar, G. Dozsa, and Alma. The deep computing messaging framework: Generalized scalable message passing on the blue gene/p supercomputer. In *Proceedings of the 22Nd Annual ICS*, ICS '08, pages 94–103, New York, NY, USA, 2008. ACM.