Collection and Analysis of Operational Traces from SURFsara Datacenters

Kristian Valur Laursen Ólason
2620524

Supervisor (VU/Leiden): dr. Alexandru Uta
Daily Supervisor (SURFsara): dr.ir. Valeriu Codreanu
Second Reader (VU): prof.dr.ir. Alexandru Iosup

July 2020
BSc thesis

Acknowledgements

I would like to thank my supervisors Alexandru Uta and Valeriu Codreanu for all of their support and Alexandru Iosup for suggesting several improvements to this paper and Laurens Versluis for the assistance with producing the graphs to represent some of my data. Additionally, I would like to thank all of my coworkers at SURF for all the help and assistance over the course of this project.

Contents

1 Introduction **1**
1.1 Research questions . . . 2
1.2 Main contribution . . . 3
1.3 Thesis outline . . . 4

2 Related Work **5**
2.1 Academic Trace Archives . . . 5
2.2 Industrial Trace Archives . . . 8
2.3 Importance of Variety in Trace Sources . . . 9

3 SURFace Archive **11**
3.1 Dataset Information . . . 11
3.2 System Overview . . . 13
3.3 Requirements . . . 14
3.4 Overview of the SURFace Archive . . . 14
3.5 Design . . . 14

4 Implementation **18**
4.1 Graphana . . . 18
4.2 Graphite and Prometheus . . . 19
4.3 Parsing Output from Prometheus. . . . 20
4.4 Slurm . . . 20
4.5 Data Cleanup . . . 21
4.6 Anonymization of the Data . . . 21
4.7 Automating the Data Collection . . . 22

5 Job Characterization **23**
5.1 The Job Data . . . 23

5.2 Characterization 26
5.3 Analysis 27

6 Conclusion **28**
6.1 Future work 29

A Metrics **32**
A.1 General Hardware Metrics 32
A.2 Memory Metrics 33
A.3 Graphical Processing Unit Metrics 34
A.4 IO metrics 35
A.5 General Networking Metrics 35
A.6 Networking Sockstat metrics 36
A.7 Networking Netstat Metrics 36

B Code-Base **40**
B.1 Required Libraries 40
B.2 Data Collection 40
B.3 JSON Parser 46

Chapter 1

Introduction

In modern computing running complex analysis and modelling is more often than not executed in cloud based datacenters which perform the computation. Some of the problems as a result of higher demand are: increased energy demand, need for upgrading hardware at a faster rate, and more construction required for new datacenters. This comes with a significant environmental cost. This demonstrates the necessity of research into methods to mitigate this cost.

Operational traces form the backbone of workload and scheduling analysis of modern datacenters. In this thesis we describe a dataset of such traces where we have collected hardware metric time series with a 15 second granularity for over 100 metrics. Those metrics include CPU, GPU, memory, disk and networking information obtained through the Intelligent Platform Management Interface (IPMI) and the Nvidia Management Library (NVML).

Traces such as these are vital because they allow systems administrators to make informed design decisions rather than making assumptions without the data to back them up, as well as allowing full stack analyses of the OS-level operation of datacenters.

Datacenters are large systems comprised of multiple computers hereafter referred to as nodes. These nodes can vary in setup and are used to compute information within a datacenter. Nodes are usually housed in racks of which there can be many in the datacenter itself. To compute information these systems use a scheduler which takes compute jobs submitted by users and automatically selects nodes or even dedicates whole racks to run the calculations needed as per the parameters of the job.

All these data were collected by the monitoring system and time series

database named Prometheus from the SURFsara HPC cluster Lisa with a release from the Cartesius cluster expected in the future. Additionally, this dataset contains job and scheduling information from the system as well as topology information and the scheduling configuration.

Traces such as these are valuable because they allow for in-depth analyses of datacenter operations. For example, power usage comparison between different clusters, partitions, nodes and jobs, as well as workload analysis and predictions, all of which can serve to reduce the environmental footprint of the datacenter with improved scheduling which lowers overall energy consumption.

Several notable trace archives have been made available for research in the past resulting in various articles including but not limited to, the Parallel Workloads Archive (PWA) [6] for analysis of parallel workloads on supercomputers. Another such archive is The Workflow Trace Archive (2019) [14] which looks into workflow traces found in datacenters, clouds and HPC systems. Large archives such as the Google [11] and Microsoft [5] collections are particularly important due to the rarity of large multi-day archives from such high profile corporations. None of the aforementioned archives have the low level metric granularity of our SURFace [9] archive. To our knowledge there does not exist a published archive with this level of detail. Our work uniquely fills this gap in modern data collections allowing for new avenues of research when it comes to the analysis of data center operations.

Our data is collected from the Dutch National Infrastructure Lisa. This infrastructure consists of 343 nodes and we collect the data by querying a monitoring system on the cluster for each of our collected measurements.

1.1 Research questions

In our work we aim to publish a dataset of considerable detail, this requires a significant amount of analysis work as well as development work to accomplish. Through this we define several research questions that we need to resolve:

1. How do we collect such a vast amount of data and what are the potential pitfalls when performing such large scale data collection? To be able to publish a dataset of this size we must first define how to collect the data and design ways to store it.

2. What kind of processes are required to automate such a data collection to facilitate future data releases? Collecting this much data necessitates the development of data collection scripts.

3. What kind of knowledge can we derive from the archive by characterizing the jobs that run on datacenter clusters?

1.2 Main contribution

In answering these questions we contribute the following to the research community at large:

1. An archive that has been made available to the public for research and is available at https://zenodo.org/record/3878143.

2. The process of collecting and processing a large amount of data from traces like these. As well as, highlights of some common pitfalls researchers can encounter.

3. An overview of the automation process and the setup required at the data center.

4. Job characterizations and analysis from a national infrastructure with an overview of how the system is used.

The archive published alongside this thesis is a significant improvement over existing archives due in part to the granularity of the data. The archive is a collection of system measurements called metrics collected from a datacenter. This dataset contains measurements at 15 second intervals per node, per metric resulting in billions of unique datapoints collected over the course of three months with releases planned on a monthly basis using automated scripts to ensure consistent data releases. Additionally it provides system information in the form of scheduling configuration, topology information, and metrics that are usually not available in other trace archives. These novel metrics include for example memory utilization, GPU power and temperature measurements. This archive alongside the work showcased in this thesis opens up important and interesting avenues for future research and may prove invaluable to datacenter planning for years to come. We also have access to unreleased job data consisting of six months of data collected from

January 2020 to July 2020. The reason we have six months of Job information is that it requires less processing than the hardware metric information to be used for analysis.

1.3 Thesis outline

The structure of the thesis is as follows. In chapter 2 we cover existing literature on this subject and show what kind of analysis already exists on data similar to what this archive contains as well as briefly discuss what other recent archives have contributed in this field. In chapter 3 we provide an overview of the system, the project requirements, and the overall design of the archive. In chapter 4 we discuss in depth the implementation of the data collection and how the problems faced during development were solved. In chapter 5 we characterize the jobs that run on the cluster to demonstrate what types of workloads can be expected. Finally, in chapter 6 we conclude our work by highlighting our findings and propose future developments.

Chapter 2

Related Work

In this chapter we discuss previous work related to our own. Additionally, we contrast their work with ours to highlight how we improve upon previous work.

As mentioned before this is by no means the first trace archive and analysis on a data-center system. A not insignificant number of archives and articles on those archives have already been published and here we present the articles most relevant to our work.

2.1 Academic Trace Archives

In this section we discuss research work on trace archives from various academic sources and contrast their work with ours.

Parallel Workloads Archive

One of the oldest repositories for workload archives is the aforementioned Parallel Workloads Archive (PWA)[1]. The archive consists of workload logs and workload models collected from systems around the world. This is meant to provide free access to data and analysis for researchers with the intent to facilitate further research into parallelism and scheduling on large computer systems.

The main goal of the PWA paper [6] which accompanies the archive is to highlight these points:

[1] https://www.cs.huji.ac.il/labs/parallel/workload/

- How to work with the logs hosted by the PWA.
- Evaluation and system design has much to gain from having access to detailed workload logs.
- The importance of data reliability when producing performance evaluations.
- Even if workload logs are automatically generated they may have data quality problems.
- Identifying such data quality problems can be difficult and a report on those problems if found is necessary.
- There are existing methods to improve data quality.

Feitelson et al. observe that data and logs such as the ones they have in their archive all have data quality problems. The authors outline the 5 most important qualities that logs such as these should have. Those are, by order of importance, accuracy, consistency, security, timeliness and completeness [8].

They conclude that even though computer science is a discipline of rapid advancement where empirical data becomes outdated early, datasets such as the PWA, and by extension our SURFace archive [9], are highly important to ensure that advancement is data driven rather than based on assumptions. Additionally, they highlight the importance of data sharing since a single user cannot be expected to analyze and detect artifacts in the data on their own. Therefore, sharing our findings is just as important as sharing the data itself.

Our work provides an improvement over existing archives in PWA due to the data granularity and detail. The diversity of the metrics we have collected as well as the collection frequency provide a better insight into how datacenter hardware performs on a daily basis.

Grid Workloads Archive

The Grid Workloads Archive (GWA)[2] is meant to expand the availability of grid workload traces and encourage collaboration of grid researchers.

[2] http://gwa.ewi.tudelft.nl/

Iosup et al. [7] focus mainly on the lack of existing publicly available grid workload archives. They also discuss that grid workloads traditionally consist of single node jobs as opposed to parallel workloads which are generally multi node jobs. Additionally, they present the application of GWA in several areas. It is worth noting that the GWA is referenced on the PWA website which is a great example of collaboration within the scientific community.

The GWA archive consists of traces from nine different grid environments. All of these grid workloads are available at the GWA website. And through their work they designed, produced and published a set of workload trace archives.

Our SURFace Archive [9] consists of a single cluster. However, the data we collect is more fine grained and is planned to be released consistently every month. This ensures that researchers will always have access to up to date data and the ability to analyse historical trends. Additionally, our focus aligns with the focus of GWA with regards to providing public access to cluster workloads and hardware metrics. This enables researchers to plan based on data rather than assumptions.

The Workflow Trace Archive

Versluis et al. [14] in their work on the Workflow Trace Archive (WTA)[3] stress the importance of diverse workflow traces. The importance of this is due to the lack of existing traces which is evident in the fact that less than 40% of the research papers they inspected are making use of realistic data in their analyses. And what is worse is that less than 15% of them are using open access traces. This has a serious impact on the reproducibility of their work.

They published the WTA archive which is a set of workflow traces from 10 different cloud systems which encompasses more than 48 million workflows. It is worth noting that the WTA has grown in size since then due to allowing organizations to contribute to the trace archive with their own data.

Versluis et al. also provide workflow characterizations in their work which highlight the difference between types of workflows such as parallelism. This emphasizes that to obtain a good overview of workflows researchers must perform analyses on traces from different sources.

To contrast, our archive focuses on one source for the data, but provides

[3] https://wta.atlarge-research.com/

a much more fine grained look at datacenter activity through hardware measurements rather than focusing on workflow data. However, once we are able to supplement our data with the job information from these systems workflow analysis becomes possible.

2.2 Industrial Trace Archives

In this section we discuss industry traces from the Google and Microsoft clusters.

Google Trace Archive

Mishra et al. [11] observe that the increased prevalence of cloud computing in modern industries necessitates good scalability in compute clusters which deal with ever increasing workloads. Their work aims to improve capacity planning and task scheduling on large scale systems.

They describe an approach to workload classification and correlate it with workload traces from the Google Cloud Backend[4]. They use a number of methods to classify the workloads and combine them into different categories to reduce the overall number of workloads on the system. This results in faster execution on the system rather than every task being assigned its own workload.

Through their work they identified that most of the tasks that run on the system have a short duration. However, most of the resources on the system are allocated to longer tasks. We have seen similar indications through our work with the SURFace archive and job accounting data from Lisa where out of over 1.9 million jobs that are executed, roughly 10 thousand of those jobs have a longer duration than one day. However, those jobs consume about 80% of the allocation time on the system.

In their work they examined 20 days of data. By contrast our work provides researchers with access to 3 months of information with more releases planned in the near future. The length of our data collection provides a multitude of benefits such as long-term trend analysis and enough diurnal and weekly data to be statistically significant.

[4] https://github.com/google/cluster-data

Microsoft Trace Archive

Cortez et al. in their work [5] perform workload characterization of Microsoft Azure VM [4]. Which includes a distribution of resource consumption. Their reasoning for their research is the lack of existing data on production Virtual Machines (VM). They aim to provide more data which with research could improve resource allocation in production systems.

Additionally, they show that VM workloads in the production systems they examined have a consistent pattern to them. This allowed them to develop software which after training offline which they claim can accurately predict the cloud workloads on these systems.

They then show that the prediction software improves performance in systems using real VM traces through improved resource allocation and utilization and reduces instances of the scheduler over-committing resources.

Through their work they published a dataset of traces collected from several large scale Microsoft Azure VM systems[5].

In our work we aim to facilitate such analysis on different types of production clusters with research workloads similar to the SURFsara systems. These different workloads enable researchers to have a more complete picture of datacenter workloads and reduced homogeneity of published datasets.

2.3 Importance of Variety in Trace Sources

In this section we briefly summarize the article by Amvrosiadis et al. [2]. In their work they state that data scarcity leads to users overfitting their expected workload types to the Google trace. Through their analysis they found that private cluster workloads resemble High Performance Computing (HPC) workloads more than they do the Google cluster workloads.

In their paper they introduce four new traces. Two of which originate in privately owned clusters and two from public HPC clusters. They perform characterization of the workloads from these newly introduced clusters and contrast it with the Google dataset. They found that there is a significant difference between the workloads on these clusters compared to the google workload. This emphasizes the importance of variance in available logs, especially when performance evaluations are taken into account.

[5] https://github.com/Azure/AzurePublicDataset

Alongside their work they published a trace archive consisting of the LANL Trinity and LANL Mustang clusters as well as the two traces recorded from the Two-Sigma system[6]. Similar to our traces this allows for further analysis into the workloads of various different systems to reduce overfitting to one type of workload.

[6]https://www.pdl.cmu.edu/ATLAS/

Chapter 3

SURFace Archive

In this chapter we describe our design of the SURFace archive published on Zenodo[1]. Including the requirements to collect the data, as well as, the design choices that were made along the way when the data was being processed.

The trace archive published alongside this thesis consists of software and hardware traces collected from Dutch national Compute Cluster, LISA, hosted at SURFsara [9]. An upcoming article in USENIX;login; [13] is expected to be released soon as a result of our work. Additionally, we are providing support for three upcoming projects. We plan to add to this archive the job data for each job that runs on the cluster.

In this and the following chapter we answer the following two research questions:

- How do we go about collecting such a vast amount of data and what are the potential pitfalls when performing such large scale data collection?
- What kind of processes are required to automate such a data collection to facilitate future data releases?

3.1 Dataset Information

We introduce hardware metric timeseries collected from a production cluster used by researchers from various research centers in the Netherlands.

The archive has numerous advantages over existing archives as shown in figure 3.1. The amount of novel data we provide in the archive as well as

[1] https://zenodo.org/record/3878143

Platform	Nodes	CPUs	RAM	GPUs per node	Length
SURFsara gold_6130	192	16	96GB	-	3 months
SURFsara silver_4110	96	16	96GB	-	
SURFsara gold_5118	29	24	192GB	4x Titan RTX	
SURFsara bronze_3104	23	12	256GB	4x GeForce 1080Ti	
SURFsara bronze_3104	2	12	256GB	4x Titan V	
SURFsara gold_6126	1	48	2TB	-	

Table 3.1: Hardware information for the Lisa system.

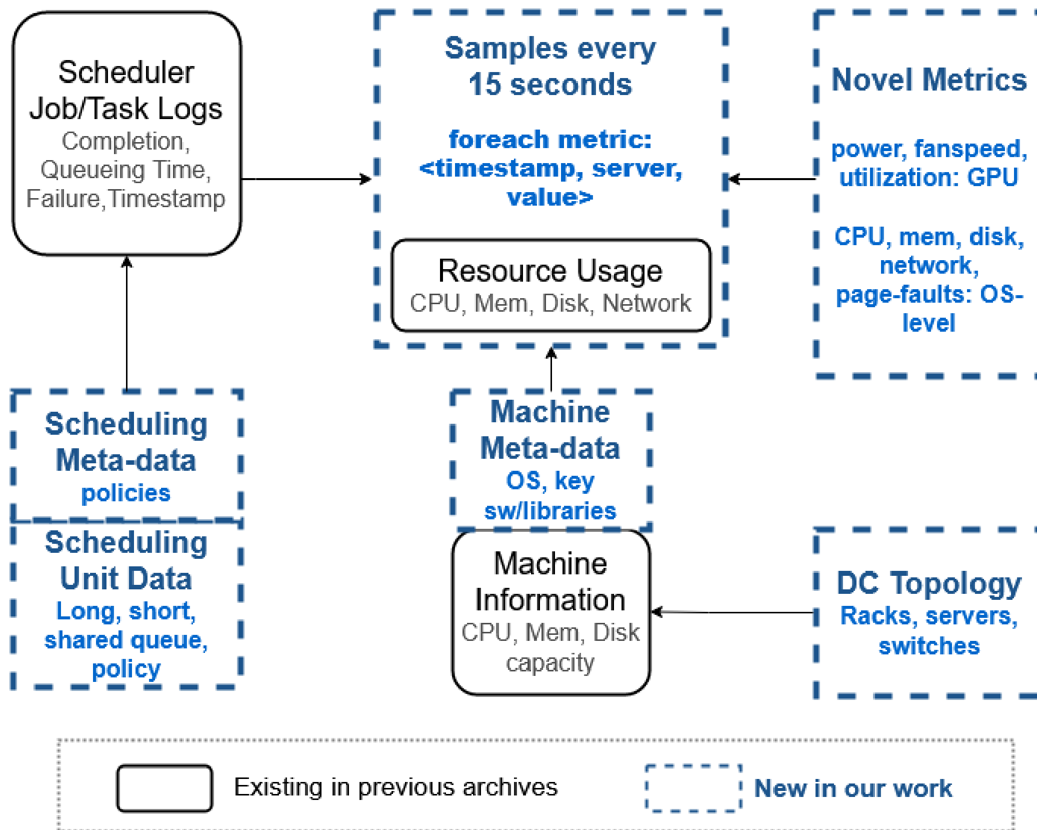


Figure 3.1: A visual representation of what we are providing with our work compared to what exists in previously released archives.

the granularity is unparalleled in the wild and opens up multiple promising avenues for future research. As shown in Figure 3.1, existing previous archives have typically contained at least one of these metrics.

1. Scheduler Job/Task Logs: Job information from the scheduler which allow for workflow analysis on the systems through analysis of jobs.

2. Resource Usage: An overview of resource usage on a high level, which enables researchers to make some assumptions of system usage.

3. Machine Information: The overall capacity of the system. This typically makes resource allocation comparable to machine information to see if a system could be over-committing resources to a particular job.

What we are introducing is an archive which contains significantly more detailed data with more measurements, greater diversity of metrics and information than comparable archives that we have looked at.

Our archive consists of 138 metric time-series collected at 15 second intervals. This means that each day of measurements consists of 5760 data points per node for each metric that is collected. To put that in perspective that is more than 750.000 data points for a GPU node in a single day.

3.2 System Overview

The Dutch National Infrastructure, Lisa consists of 343 nodes. Of which there are 54 GPU nodes and 289 compute nodes[2] 3.1. The system handles primarily research based workloads from research institutions accross the Netherlands.

The metric data is collected by system and saved in short term storage by the time-series monitoring system Prometheus[3] [3]. Prometheus stores time series based data collected from software daemons hosted on each node. These daemons transmit sensor data from different systems on the node to a central server node that stores the data.

Job scheduling is handled by the SLURM Workload Manager [10] and stored in the accompanying accounting software Slurm Accounting (Sacct). Sacct stores various job information from the system.

[2] https://userinfo.surfsara.nl/systems/lisa/description

[3] https://github.com/prometheus

3.3 Requirements

Collecting such data from a professional, production cluster requires preparation, access rights, and proper data-collection technology. For this particular archive the author was employed as an intern to work on the project at SURFsara from September 2019 until July 2020, because the data is only accessible from within the system itself. Additionally, this project requires a basic understanding of using HTTP requests.

In our work we made use of Python scripts to perform the data collection and parsing. The libraries required to run the scripts can be found in the code base appendix under Required Libraries B.1.

3.4 Overview of the SURFace Archive

When designing the SURFace archive we had to take our stakeholders into account. The data provider SURFsara is our *contributor*. They contribute the data to the project and provide access to the system which allows us to collect the data. Our trace contains the metric measurements from the Lisa cluster. This includes resource usage, resource descriptions and job data. We work with SURFsara in designing the processes to perform the collection and anonymization of the data and publication. Additionally, we work on the analysis and research of the data post collection.

The potential *consumers* of the data are split into two stakeholders. *Systems admins* can use the data we collect to predict the workloads of the clusters they oversee as well as decide on upgrades for other existing clusters. *Researchers* have a vested interest in exploring the user behavior on such systems and how well resources are allocated to develop schedulers and find ways to optimize power usage in production clusters.

3.5 Design

In this section we explain some of our design choices made during when working on this project.

Data Collection

As previously mentioned our data collection and conversion process is handled by Python scripts. We chose Python due to the simplicity of implementation. Languages such as C++ or Java would have required a much bigger time investment in designing the required frameworks to facilitate data collection. The Python scripts are developed by the author of this thesis with support from the project supervisors and employees at SURFsara and they are responsible for maintenance and improvement of these scripts. The collection script queries the Prometheus database for the data. Additionally it queries the Sacct database for the job data. This data collection runs on a daily basis on the system.

Storage Format

We use Parquet [15] as our chosen storage format due to the fact that it takes up a minimal amount of space. This storage format also has native Spark [12] support and is also readable by Tensorflow [1] without additional libraries.

The output format from the monitoring software on the Lisa cluster is in complex Java-script Object Notation (JSON) which is purely in string format and as such has suboptimal space complexity. Therefore, we needed to come up with a better storage format which takes up less space while also allowing for fast look-ups. Among those we considered were using an SQL database which while better than JSON still proves inadequate for such a large time-series due to the amount of tables and rows per table. CSV, while good for machine learning, shares the problem JSON has with regard to storage space.

We converged on Parquet which is a columnar storage format with built in compression which reduces the space complexity by a significant amount. Parquet is also capable of interfacing with Spark-SQL which is an analytics engine for large-scale data processing. Another motivation behind Parquet it that the Tensorflow library has native support for it making it a suitable format to use in conjunction with machine learning frameworks for data analysis. Parquet was designed by Apache to be an efficient columnar storage format with support for multiple types of compression algorithms and capability of storing large blocks of data with minimal space complexity.

Job ID	Unique ID number of the job
Group ID	Unique ID of the User group the submitter is in
User ID	Unique ID of the User
Partition	The Partition the user selected for the job
Submit	Submission time in seconds since epoch
Start	Start time of the job in seconds since epoch
End	End time of the job in seconds since epoch
Elapsed raw	Job runtime in seconds
CPU time raw	CPU time in seconds
Number of CPU's	Number of CPU's allocated to the job
Number of nodes	Number of nodes allocated to the job
Node list	A list of the nodes allocated to the job
Exit code	The exit code[4] of the program
State	End state of the job
Timelimit	User selected time limit. (Default 5 min.)

Table 3.2: List of the sacct output we query.

Slurm

The Slurm data is queried in the format readable from Table 3.2. This data is then parsed and the GroupID gets removed, the Job ID and User ID's get encrypted as well as the nodelist. Due to GDPR concerns as of now this data is not published alongside the archive. We do plan to release this data once these concerns have been addressed or our anonymization has been deemed sufficient.

Data Anonymization

Our collection of job data comes with some inherent problems. Some of the information contained in the data could allow users of the system to potentially identify other users which can have serious implications for SURFsara and our research team if identifiable. Therefore we were faced with the question. How can we ensure that user information cannot be obtained from our data without destroying the data itself?

Our data does contain some identifying characteristics which necessitate methods of anonymization such as hashing and encrypting those characteristics. This is a difficult problem to solve and we are currently further

investigating methods of ensuring user anonymity. We further discuss this in chapter 4, section 4.6.

Data cleanup

After the data collection we noted that some metrics in the collection never return a value or consistently always return a single unchanging value. This could mean that the monitoring for that particular metric could be disabled or the sensors for those sensors are not working. This necessitated the development of a parsing script to filter out those metrics. This is further discussed in chapter 4, section 4.5.

Automating the Data Collection

We are working toward the development of an automated script that will facilitate the daily data collection and monthly data uploads to our Zenodo repository. This ensures that no data is lost and that the data is uploaded on a regular schedule to the repository to provide consistent data releases for future research. This is further discussed in chapter 4, section 4.7

Chapter 4

Implementation

In this chapter we discuss the implementation of the project, some of the problems encountered and the design decisions made during the development process. We start by going over the various systems we explored starting with Graphana. Followed by the time-series systems Graphite and Prometheus. After this we discuss how we parse the output from Prometheus followed by an explanation of the job data collection. Finally we go into the data cleanup, anonymization and collection automation. We conclude the chapter with a high level explanation of our data collection process.

During the implementation of the data collection project we encountered several problems and design issues we had to address as well as setbacks we had to work past. At the start of this project the goal was to extract data from the Cartesius cluster followed by the Lisa cluster. After identifying potential means to collect the data and gaining access to those systems we encountered several problems that needed to be addressed. We discuss these problems in the following sections.

4.1 Graphana

The first few weeks of the project were spent on building an understanding of the systems at SURFsara and finding out how they process their internal data. To visualize the data SURFsara uses the Grafana Analytic Platform and queries time-series databases to enable graphing of the data. We deemed this platform to be promising as it was already visualizing the data we wanted to collect. However, after obtaining elevated access privileges it soon became

apparent that there was no accessible API to retrieve data from Graphana.

After some inquiries we were informed that this is not a planned feature of the platform. We did some research into attempting to web scrape the Graphana interface for the data but we quickly abandoned that due to the complexity of such a project would amount to a new research project on its own. The only way to collect the data from Graphana would be to manually download the information every day with multiple clicks per data point so a decision was made to drop any further attempts to collect data from this source.

4.2 Graphite and Prometheus

After the lack of success with our work on with Graphana we moved onto the time-series databases it was querying. The platform used for Cartesius, called Graphite, is a monitoring tool for hardware and cloud infrastructure. Graphite stores time-series data that it receives from the Baseboard Management Controller (BMC) which is configured to push sensor information once every 10 minutes. Graphite has a HTTP API accessible through terminal requests from a system with administrative access, however, due to delays in receiving credentials to allow for querying we moved onto the Lisa cluster.

The Lisa cluster makes use of a robust time series database called Prometheus. Prometheus is open source and uses PromQL which is a query language that allows for a multitude of complex queries to access the data stored in the database. Prometheus works differently from Graphite in a way that it scrapes data from software daemons running on each node. Notably Lisa makes use of node exporter[1] and nvidia gpu prometheus exporter[2] to facilitate the data monitoring.

We acquired access to an administrative node to begin our data collection testing on Lisa and began preliminary analysis and collection using curl scripts to construct and collect some data. Shortly after that we modified how the data is accessed by using a special node on the Lisa system so that data collection could be performed without requiring access to the systems admin node for collection.

[1] https://github.com/prometheus/node_exporter

[2] https://github.com/mindprince/nvidia_gpu_prometheus_exporter

4.3 Parsing Output from Prometheus.

The output from Prometheus is in complex JSON format. What we mean by complex is that it is a nested data structure for each metric, which contains a large array of data structures that consist of node information as well as an array of time series for each node. The measurements collected are stored as a tuple of time and value. To be able to access this data we developed scripts in Python 3.7 that use several packages to interface with the JSON data on a daily basis. These packages are can be found in the appendix at B.1:

1. json: Exposes an api that allows the user to interface with JSON formatted data.

2. Pandas: an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

3. pyArrow: provides a Python API for functionality provided by the Arrow C++ libraries, along with tools for Arrow integration and interoperability with pandas, NumPy, and other software in the Python ecosystem.

4. pyArrow.parquet: Enables python bindings to multi threaded C++ code that allows for reading and writing of parquet files.

Using the json library we are able to interface with the Prometheus data and for each result we extract in a linear fashion each node as well as the accompanying time series data. We decouple the data tuples and then insert the data into a Pandas dataframe, which is a sort of column based data structure, where the index column is the time data in seconds since epoch and each column represents a day of data collected for a node. Subsequently, we make use of the pyArrow library to convert the pandas dataframe to a pyArrow table which enables the use of the pyArrow Parquet library to save each metric in Parquet format.

4.4 Slurm

The Lisa and Cartesius clusters use the Slurm workload manager which is a highly scalable cluster management and job scheduling system for Linux

clusters. Slurm handles the job scheduling on these systems and can be configured in a multitude of ways to improve scheduling and the occupancy of the compute clusters. It is worth noting that on both of the clusters the default time limit for each job is 5 minutes. The users of the system can set a longer time limit with a cap of 5 days on most partitions. The job data is then stored in the Slurm accounting database which enables access to various historical data on the jobs that have run on the cluster. The most relevant of those we have found to be:

1. Job submit, start, and end time.
2. The timelimit on a per job basis.
3. Number of nodes reserved as well as a list of nodes by name.
4. the state of the job, such as, completed, failed, cancelled and timeout.
5. Exit code of the jobs, which may provide information on failed jobs if they, for example, have a segmentation fault.

Included in our data collection script is code that queries Slurm for job accounting data to allow for more robust analysis of the operational traces on the system. These job data can be instrumental in detecting ways to improve scheduling and energy efficiency in datacenters as well as researching system and user behavior on large compute clusters.

4.5 Data Cleanup

We developed a simple parsing script that ran through each metric and identified the problematic ones and removed them from the dataset as well as stopped querying for those metrics from the Prometheus system. This reduced the number of initial metrics from 191 to 133 which saves a considerable amount of time during the data collection as well as disk space.

4.6 Anonymization of the Data

To better facilitate analysis it is useful to be able to tell if a certain user on the system is uploading multiple jobs with a similar hardware signature. Our

first approach was to attempt to encrypt the user id's and the accompanying job id's to prevent other users of the system from looking them up through the Linux user base and the Slurm scheduler. We also encrypted our node names to make it harder.

However because these data contain the submission, start, and end times of jobs this proves difficult since rare occurrences on the system would be easily identifiable, such as very large jobs that reserve most of the cluster. Even non significant events could be identified due to the fact that it is highly unlikely that two jobs share the same submit, start, and end times for each of those values. From that information anyone with access to the system could discover the user behind each individual job in the dataset.

Obfuscating those time values would effectively destroy the usability of the job data. Due to this we have had to delay the release of the job data with our current archive. However, discussions are under way to find a solution to this problem or at least, in part, release some job data overview such as how many jobs are running at any one time and which partitions they are running on, which nodes are reserved and which are idle.

Work is currently ongoing to see if a release of job information is possible some time in the future.

4.7 Automating the Data Collection

We are working on an automatic script that will upload an archive to Zenodo monthly. We achieve this by running automated processes on an administrative node within the Lisa cluster. Uploading to the Zenodo archive requries the use of an API they have set up on their end. This helps to ensure regular data releases as well as reliability in a monthly release and facilitates future research and experiments in relation to datacenter operations.

Chapter 5

Job Characterization

In this chapter we characterize the kinds of jobs that run on the Lisa cluster. The data we are working with is a collection of 1,927,420 jobs that ran on the cluster and were collected over a 6 month period. We aim to replicate some of the characterizations demonstrated by Amvrodiasis et. al. [2]. We conclude the chapter by showing the common types of jobs that run on the system. As well as, a comparison of our results to the other paper.

5.1 The Job Data

The data we use consists of six months of job data collected from the Dutch National Supercomputer, Lisa. This job data consists of 1.927.420 jobs and was recorded from 1st of January 2020 until 30th of June 2020. This allows us to perform analysis on job trends within the cluster. When looking at the job data we will consider the following information:

1. Job Trends: We show the submissions per day accross the 6 months.
2. Partition: The partition the job ran on.
3. Times: The submission, start and end time of jobs.
4. Runtime: The amount of system time occupied by a job
5. State: Which displays the final state of the job, such as completed, cancelled, failed.

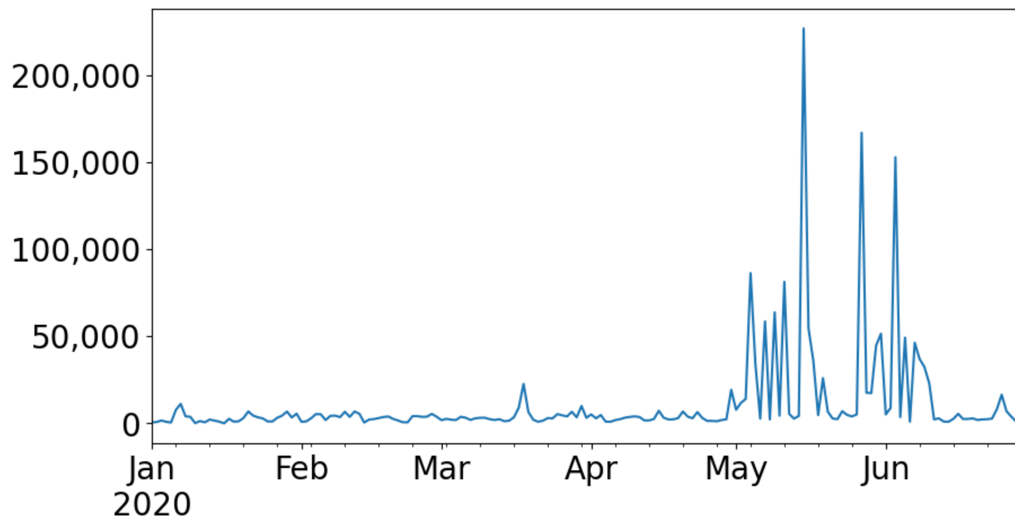


Figure 5.1: Job submission trends from 1st of January until the 30th of June.

Analysis of each of these metrics separately and combined gives a good picture of user behavior and job trends on the cluster.

Figure 5.1 shows the submission trends on the system which are relatively stable over the whole dataset. However, there are noticeable peaks in submission rates in the months of May and June. These upticks correlate with a contract on data analysis with relation to a research contract with scientist performing analysis of the Covid-19 genome in an effort to come up with methods to treat the virus. To make better sense of the submissions per day we make a boxplot of the averages per day. As can be seen in Figure 5.2 when one considers the left figure you can see the whiskers are catching the largest outliers and those outliers fall on Monday, Wednesday and Friday. As is apparent these outliers are in the 100's of thousands of jobs with Friday being the largest outlier. When you inspect the zoomed in figure you can see that on weekdays the average job submissions lie between 3-5000 jobs per day, with weekends dipping down to around 2,500 jobs per day.

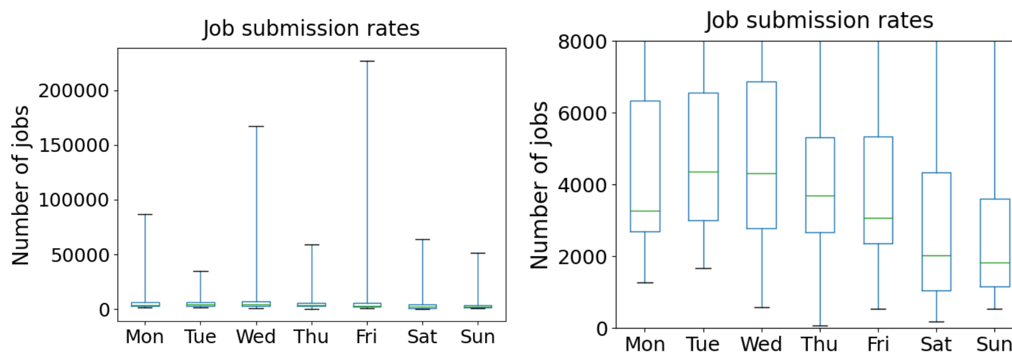
As for the jobs we collected. The partitions they ran on as well as the number of jobs per partition can be seen in Table 5.1.

Partition Name	Nr of jobs
normal	1,844,618
gpu_shared	29,484
gpu_titanrtx_shared	10,901
gpu_short	10,328
gpu_titanrtx	7,121
gpu_shared_course	5,583
gpu	5,486
short	4,457
sw	3,463
shared	1,996
Other	3983

Table 5.1: Number of jobs assigned to each partition.

End state	Count	Ratio
Total	1,927,420	100%
Completed	1,739,887	90.27%
Failed	100,455	5.21%
Cancelled	65,338	3.39%
Timeout	20,700	1.07%
Out of memory	864	0.048%
Node failure	176	0.00903%

Table 5.2: End states of all jobs running from January to July.



(a) Full representation of job submission rates per day.

(b) Zoomed in image of job submission rates.

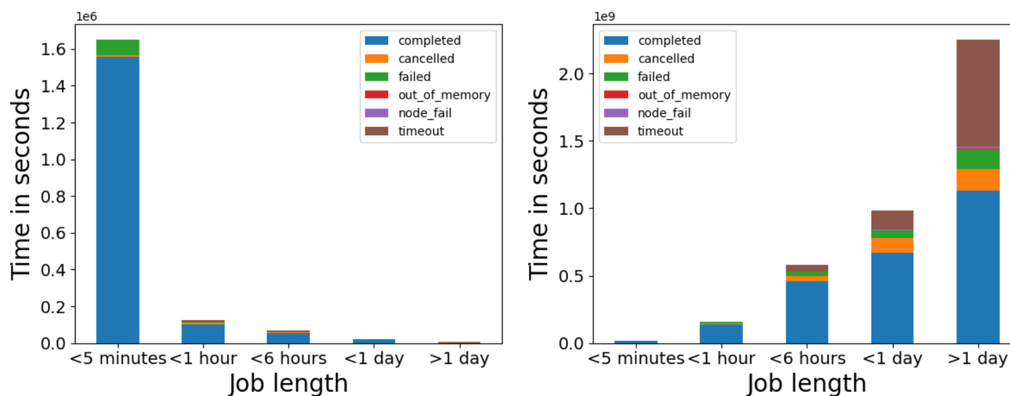
Figure 5.2: Plots that show submissions per day. The whiskers represent outliers and have been set to capture all of them. The second image shows a zoomed in version of the first. This allows the reader to get a better sense of the normal values of job submissions per day

5.2 Characterization

Table 5.2 shows that most jobs on the system complete without problems. Additionally, users of the system tend to have good intuition as to the timeout should be. Node failures and out of memory errors are exceedingly rare as well which tells us the system is very stable.

When comparing tables 5.1 and 5.2 we can see that 95.7% of jobs run on the normal partition and that 90.27% of jobs successfully complete. It is therefore apparent that categorization of jobs in this way does not get us very far so additional categorization is required. We added a categorization of the jobs by their runtime as well as the final end state of the jobs to provide us with a clearer picture of what kind of jobs run on the Lisa cluster. This resulted in Figure 5.3. The categorization is as follows: Jobs that run for five minutes or less, one hour or less, six hours or less, one day or less, more than one day.

When we consider the plots we can see that 85.7% of jobs have a runtime of less than 5 minutes and out of those 5.6% of them fail. By contrast if we consider the longest jobs on the system they account for 0.53% of the jobs on the system however when looking at the the right graph we can tell that most of the runtime on the system is taken up by those jobs, and 46.6%



(a) Number of jobs per category. (b) Runtime of jobs per category.

Figure 5.3: Plots (a) and (b) show a categorized version of the job counts and the job runtime respectively.

of those jobs fail due to timeouts, crashes, node failures, memory errors or cancellations. This follows the intuition that the longer jobs run the more likely they are to fail due to crashes, system failures or timeouts.

5.3 Analysis

We find that if we categorize jobs by runtime almost all jobs that run on the system are short jobs that run for 5 minutes or less. However most of the system occupancy comes from the longest lasting jobs which last for one day or longer. Additionally, it follows from intuition that longer jobs are the jobs most likely to fail due to timeout, system failure, out of memory or other problems.

The most used partition on the system is the normal partition, the reason for this cannot be ascertained but intuitively this may be because users do not understand how to request partitions on the system. The number of users utilizing the shared GPU partition is promising. This means the GPU partitions are having their intended effect of sharing the load on the system between jobs that do not need full node utilization.

Chapter 6

Conclusion

We collected and published the SURFace trace archive. Alongside the trace archive we also published an article in USENIX;login [13]. This archive is already proving to be a valuable resource for systems research by the @Large research group as well as internally at SURFsara with planned submissions to well-known conferences in the near future. We also collected and worked on anonymizing job data. This data is slated for release in the future once privacy concerns have been addressed and the anonymization is deemed sufficient.

At the start of this thesis we highlighted several research questions. We answered them as follows:

RQ1: We developed processes and scripts to facilitate a large scale data collection. These processes required the development of scripts and data parsing methods to store and work on the data.

RQ2: We found out that to automate data collection processes we need the assistance of systems administrators to run automated data collection jobs on a daily basis.

RQ3: To characterize the jobs we first need to categorize them by runtime to be able to have a good idea of what the most common type of job is on the system and follow that up by categorizing by job end state for example: completed, failed and timeout.

6.1 Future work

In the future a planned data collection will be released from the Dutch National Supercomputer, Cartesius. Due to lack of resources the data collection from the Cartesius systems is outside the scope of this project and as such it was left for future research. Additionally, each of the categories from the metrics A has the potential for further analysis on HPC systems and resource management. Our work can still be improved upon with more robust metric validation. The collection and storage of the data as well as the processes used for to parse the data could still be improved.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] George Amvrosiadis, Jun Woo Park, Gregory R Ganger, Garth A Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *2018 {USENIX} Annual Technical Conference ({USENIX} {ATC} 18)*, pages 533–546, 2018.

[3] Brian Brazil. *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring.* " O'Reilly Media, Inc.", 2018.

[4] Michael Collier and Robin Shahan. *Microsoft Azure Essentials-Fundamentals of Azure.* Microsoft Press, 2015.

[5] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.

[6] Dror G Feitelson, Dan Tsafrir, and David Krakov. Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 74(10):2967–2982, 2014.

[7] Alexandru Iosup, Hui Li, Mathieu Jan, Shanny Anoep, Catalin Dumitrescu, Lex Wolters, and Dick HJ Epema. The grid workloads archive. *Future Generation Computer Systems*, 24(7):672–686, 2008.

[8] Shirlee-ann Knight and Janice Burn. Developing a framework for assessing information quality on the world wide web. *Informing Science*, 8, 2005.

[9] Kristian Valur Laursen Ólason, Alexandru Uta, Alexandru Iosup, Paul Melis, Damian Podareanu, and Valeriu Codreanu. Beneath the SURFace: An MRI-like View into the Life of a 21st Century Datacenter, June 2020.

[10] SchedMD LLC. Slurm schedmd, 2011-2019.

[11] Asit K Mishra, Joseph L Hellerstein, Walfredo Cirne, and Chita R Das. Towards characterizing cloud backend workloads: insights from google compute clusters. *ACM SIGMETRICS Performance Evaluation Review*, 37(4):34–41, 2010.

[12] Apache Spark. Apache spark. *Retrieved January*, 17:2018, 2018.

[13] Alexandru Uta, Kristian Valur Laursen Ólason, Alexandru Iosup, Paul Melis, Damian Podareanu, and Valeriu Codreanu. Beneath the surface: An mri-like view into the life of a 21st century datacenter, July 2020.

[14] Laurens Versluis, Roland Mathá, Sacheendra Talluri, Tim Hegeman, Radu Prodan, Ewa Deelman, and Alexandru Iosup. The workflow trace archive: Open-access data from public and private computing infrastructures. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2170–2184, 2020.

[15] Deepak Vohra. Apache parquet. In *Practical Hadoop Ecosystem*, pages 325–335. Springer, 2016.

Appendix A

Metrics

The following contains a list of metrics collected during this project.

A.1 General Hardware Metrics

- surfsara_power_usage
- surfsara_ambient_temp
- node_context_switches
- node_intr
- node_entropy_available_bits
- node_load1
- node_load5
- node_load15
- node_procs_blocked
- node_procs_running
- node_forks

A.2 Memory Metrics

- ((node_memory_MemTotal_bytes%20-%20node_memory_MemFree_bytes%20-%20node_memory_Cached_bytes)%20%2F%20(node_memory_MemTotal_bytes))%20*%20100
- ((node_memory_SwapTotal_bytes%20-%20node_memory_SwapFree_bytes)%20%2F%20(node_
- node_memory_MemTotal_bytes
- node_time%20-%20node_boot_time_bytes
- node_memory_PageTables_bytes
- node_memory_SwapCached_bytes
- node_memory_Slab_bytes
- node_memory_Cached_bytes
- node_memory_Buffers_bytes
- node_memory_MemFree_bytes
- node_memory_Committed_AS_bytes
- node_memory_Mapped_bytes
- node_memory_Active_bytes
- node_memory_Inactive_bytes
- node_memory_CommitLimit_bytes
- node_memory_Dirty_bytes
- node_memory_HardwareCorrupted_bytes
- node_memory_Unevictable_bytes
- node_memory_SwapTotal_bytes%20-%20node_memory_SwapFree_bytes
- node_memory_MemTotal_bytes%20-%20node_memory_MemFree_bytes%20-%20node_memory_Buffers_bytes%20-%20node_memory_Cached_bytes%20-%20node_memory_Slab_bytes%20-%20node_memory_PageTables_bytes%20-%20node_memory_SwapCached_bytes

- node_memory_Shmem_bytes
- node_memory_Mlocked_bytes
- node_vmstat_pswpin
- node_vmstat_pswpout
- node_memory_Writeback_bytes
- node_memory_Active_file_bytes
- node_memory_Inactive_file_bytes
- node_memory_VmallocChunk_bytes
- node_memory_VmallocUsed_bytes
- node_memory_DirectMap1G_bytes
- node_memory_DirectMap2M_bytes
- node_memory_DirectMap4k_bytes
- node_memory_AnonHugePages_bytes
- node_memory_AnonPages_bytes
- node_memory_Active_anon_bytes
- node_memory_SReclaimable_bytes
- node_memory_SUnreclaim_bytes
- node_memory_NFS_Unstable_bytes

A.3 Graphical Processing Unit Metrics

Node that gpu metrics come in pairs of four per node from the system. To save on storage space we opted to combine them into one column in the dataset by packing them into a 64 bit integer using a bit packing method where each value takes up 16 bits of space.

- nvidia_gpu_temperature_celsius
- nvidia_gpu_power_usage_milliwatts
- nvidia_gpu_memory_used_bytes
- nvidia_gpu_fanspeed_percent

A.4 IO metrics

- node_disk_bytes_read
- node_disk_bytes_written
- node_disk_reads_completed
- node_disk_writes_completed
- node_disk_read_time_ms
- node_disk_write_time_ms
- node_disk_io_time_weighted
- node_disk_io_time_ms
- node_filefd_maximum
- node_filefd_allocated

A.5 General Networking Metrics

- node_network_receive_packets
- node_network_transmit_packets
- node_network_receive_errs
- node_network_receive_drop
- node_network_transmit_drop
- node_network_receive_multicast

A.6 Networking Sockstat metrics

- node_sockstat_TCP_alloc
- node_sockstat_TCP_inuse
- node_sockstat_TCP_mem
- node_sockstat_TCP_mem_bytes
- node_sockstat_TCP_orphan
- node_sockstat_TCP_tw
- node_sockstat_UDP_inuse
- node_sockstat_UDP_mem
- node_sockstat_UDP_mem_bytes
- node_sockstat_sockets_used
- node_sockstat_RAW_inuse

A.7 Networking Netstat Metrics

IP

- node_netstat_Ip_InHdrErrors
- node_netstat_Ip_OutDiscards
- node_netstat_Ip_OutNoRoutes
- node_netstat_Ip_InReceives
- node_netstat_Ip_InDelivers
- node_netstat_Ip_OutRequests

TCP

- node_netstat_Tcp_CurrEstab
- node_netstat_Tcp_InErrs
- node_netstat_Tcp_InSegs
- node_netstat_Tcp_OutRsts
- node_netstat_Tcp_OutSegs
- node_netstat_Tcp_ActiveOpens
- node_netstat_Tcp_AttemptFails
- node_netstat_Tcp_EstabResets
- node_netstat_Tcp_PassiveOpens

TCP EXT

- node_netstat_TcpExt_TCPAbortOnClose
- node_netstat_TcpExt_TCPAbortOnData
- node_netstat_TcpExt_TCPAbortOnMemory
- node_netstat_TcpExt_TCPAbortOnTimeout
- node_netstat_TcpExt_TCPAbortFailed
- node_netstat_TcpExt_TCPTimeouts
- node_netstat_TcpExt_DelayedACKLocked
- node_netstat_TcpExt_DelayedACKLost
- node_netstat_TcpExt_DelayedACKs
- node_netstat_TcpExt_TCPForwardRetrans
- node_netstat_TcpExt_TCPSlowStartRetrans

- node_netstat_TcpExt_TCPSynRetrans
- node_netstat_TcpExt_TCPSpuriousRTOs
- node_netstat_TcpExt_TCPSpuriousRtxHostQueues
- node_netstat_TcpExt_TCPFullUndo
- node_netstat_TcpExt_TW
- node_netstat_TcpExt_TWKilled
- node_netstat_TcpExt_TWRecycled
- node_netstat_TcpExt_TCPDSACKIgnoredOld
- node_netstat_TcpExt_TCPDSACKOfoRecv
- node_netstat_TcpExt_TCPDSACKOfoSent
- node_netstat_TcpExt_TCPDSACKOldSent
- node_netstat_TcpExt_TCPDSACKRecv
- node_netstat_TcpExt_TCPDSACKUndo
- node_netstat_TcpExt_TCPDSACKIgnoredNoUndo
- node_netstat_TcpExt_TCPSackRecovery
- node_netstat_TcpExt_TCPSackRecoveryFail
- node_netstat_TcpExt_TCPSackShiftFallback
- node_netstat_TcpExt_TCPSackShifted
- node_netstat_TcpExt_TCPFastOpenCookieReqd
- node_netstat_TcpExt_TCPFastRetrans
- node_netstat_TcpExt_TCPHPAcks
- node_netstat_TcpExt_TCPHPHits
- node_netstat_TcpExt_TCPAutoCorking

UDP and ICMP

- node_netstat_Udp_InDatagrams
- node_netstat_Udp_OutDatagrams
- node_netstat_Icmp_InMsgs
- node_netstat_Icmp_OutMsgs
- node_netstat_Icmp_InErrors
- node_netstat_Icmp_InTimeExcds

Appendix B

Code-Base

This section contains the code base for the project.

Keep in mind due to privacy reasons parts of the scripts have been removed.

B.1 Required Libraries

- PyArrow
- Pyarrow - parquet
- Pandas
- OS
- JSON

B.2 Data Collection

```
1 import os
2 import json
3 import numpy as np
4 import time
5 import threading
6 import queue
7 import sys
8 from datetime import datetime
```

```
9  from subprocess import Popen
10 from subprocess import PIPE
11
12 """ Constants """
13 DATA_PATH = "/project/kristian/"
14 QUERY_LIST = "querylist"
15 #this is the start of the curl query that is constructed in
       the buildQuery(query) function
16 CURL_START = "curl -s '145.101.32.28:9090/api/v1/query_range?
       query="
17 #this is the time interval of the collected datapoints
18 QUERY_RESOLUTION = "&step=15'"
19
20
21 #for each key value pair in dictionary append to queue
22 def fillQueue(dict):
23     for key,val in dict.items():
24         q.put(val)
25     q.join()
26
27 #while queue has jobs perform jobs, this is thread safe
28 def worker():
29     while True:
30         item = q.get()
31         if item is None:
32             break
33         runLoop(item)
34         q.task_done()
35
36 #populate the list of workers
37 def generateWorkers():
38     for i in range(num_worker_threads):
39         t = threading.Thread(target=worker)
40         t.start()
41         threads.append(t)
42
43 #depopulate the list of workers
44 def terminateWorkers():
45     for i in range(num_worker_threads):
46         q.put(None)
47     for t in threads:
48         t.join()
49
50 #parse file for queries
51 def loadFile(fileName):
```

```
52     counter = 0
53     d = {}
54     with open(fileName) as f:
55         for line in f:
56             d[counter] = line.strip('\n')
57             counter = counter + 1
58     return d
59 
60 #build curl query
61 def buildQuery(query):
62     #this accesses the address where prometheus is.
63     queryMetric = query
64     queryStart = "&start="
65     queryEnd = "&end="
66     #queryRes = "&step=15'"
67     return CURL_START + queryMetric + queryStart + str(start)
       + queryEnd + str(end) + QUERY_RESOLUTION
68 
69 def runLoop(queryName):
70     print('query: ' + queryName)
71 
72     query = buildQuery(queryName)
73     result = os.popen(query).read()
74     jsonData = json.loads(result)
75 
76 
77     os.mkdir(DATA_PATH + "logs/" + queryName)
78     with open(DATA_PATH + "logs/" + queryName + "/" +
    queryName, 'w') as outfile:
79         json.dump(jsonData, outfile)
80 
81 
82 
83 
84 """
    ----------------------------------------------------------------------------
     """
85 
86 """ Get relevant time values """
87 #TODO::upgrade this to use UTC to avoid daylight savings
    problems
88 def getTime():
89     now = datetime.now()
90     return int(time.mktime(datetime.timetuple(datetime(now.
    year, now.month, now.day))))
```

```
91
92  def epoch_to_utc(epoch):
93      '''Convert epoch timestamp to ISO UTC timestamp'''
94      return time.strftime('%Y-%m-%dT%H:%M:%S', time.gmtime(
    float(epoch)))
95
96  """
    ---------------------------------------------------------------------------------
    """
97
98  """ Collect sacct information """
99  class Sacct():
100
101     def __init__(self, start, end, values, states):
102         self._start = start
103         self._end = end
104         self._format = values
105         self._states = states
106         self._sacct()
107
108     def _sacct(self):
109         '''Collect output from sacct'''
110         sacct_command = '/usr/bin/sacct --allocations --
    allusers --format %s --noheader --parsable2 --state=%s --
    start=%s --end=%s' % (
111                                                           ','.join
    (self._format), ','.join(self._states), self._start, self.
    _end)
112         if debug:
113             print("Sacct Command:\n" + sacct_command)
114         # use UTC timestamps as input and get epoch
    timestamps in output
115         stdout, stderr = Popen(sacct_command, shell=True,
    stdout=PIPE, stderr=PIPE, env={'SLURM_TIME_FORMAT': '%s',
    'TZ': 'UTC'}).communicate(input=None)
116         if stderr:
117             print(stderr, file=sys.stderr)
118             sys.exit(1)
119         self._sacct_output = stdout.decode().splitlines()
120
121     def get_jobs(self):
122         '''Parse sacct output and return iterator'''
123         for line in self._sacct_output:
124             if debug:
125                 print('DEBUG: sacct_output [%s]' % line)
```

```
126             sacct_fields = line.split('|')
127             if len(sacct_fields) != len(self._format):
128                 print('ERROR: sacct output does not match
    format', file=sys.stderr)
129                 sys.exit(1)
130             yield line
131
132
133 def getAndSaveSacctData():
134     sacct_values = ('jobid', 'gid', 'uid', 'partition', '
    submit', 'start', 'end', 'elapsedraw',
135                         'cputimeraw', 'ncpus', 'nnodes', '
    nodelist',
136                         'exitcode', 'state', 'timelimit')
137
138     sacct_states = ('CANCELLED', 'COMPLETED', 'FAILED', '
    NODE_FAIL', 'OUT_OF_MEMORY', 'PREEMPTED', 'TIMEOUT')
139
140     test_sacct = Sacct(epoch_to_utc(start), epoch_to_utc(
    start + CONST_DAY), sacct_values, sacct_states)
141
142     job_list = list()
143
144     for line in test_sacct.get_jobs():
145         job_list.append(line)
146
147     with open(DATA_PATH + "sacct/sacct_" + str(start) + "-" +
     str(start + CONST_DAY), 'w') as outfile:
148         for line in job_list:
149             outfile.write(line + '\n')
150
151     print("Sacct data collected")
152
153 """
    ------------------------------------------------------------------------------------------------
     """
154
155 """ Compress collected json data """
156 def tar():
157     os.system("tar czf " + DATA_PATH + "archives/logs_" + str
    (start) + "_" + str(end) + ".tar.gz " + DATA_PATH + "logs/
    ")
158     print("tar successful")
159
160 """ post run cleanup """
```

```
161 # delete all collected json file post archiving them with tar
162 def cleanup():
163     os.chdir(DATA_PATH + "logs/")
164     os.system("rm -r *")
165     os.chdir("../datascript/")
166     print("Cleanup complete")
167
168 """
    -------------------------------------------------------------------------------
     """
169
170 """ Setup """
171 # load queries into dictionary
172 q = queue.Queue()
173 threads = []
174 num_worker_threads = 1
175
176 # Getting start and end times using the getTime function
177 # then generate the indexes required for this day to be used
    in the conversion to parquet
178 midnight = getTime()
179 CONST_DAY = 86400
180
181 # the number 1 stands for the number of days back in time you
    're collecting
182 start = midnight - CONST_DAY * 1
183 # 15 stands for number of seconds, this avoids overlap with
    the next day.
184 end = start + CONST_DAY - 15
185
186
187 # Generates a list of range from start to end on a step of 15
     to match with the collected data
188 indexes = np.arange(start, end + 15, 15).tolist()
189 debug = False
190
191 """ Generate workers and fill up the queue to start the data
    collection """
192 generateWorkers()
193 fillQueue(loadFile(QUERY_LIST))
194 terminateWorkers()
195 getAndSaveSacctData()
196
197 print("Data collection complete")
198
```

```
199 """ Compress then cleanup collected data """
200 tar()
201 cleanup()
```

B.3 JSON Parser

```
1  import json
2  import pandas as pd
3  import pyarrow.parquet as pq
4  import pyarrow as pa
5  from collections import OrderedDict
6  # [Redacted encryption libraries]
7
8  # for encryption purposes
9  # [Redacted encryption keys]
10
11 # Constants:
12 NODE_LIST = 'nodeList'
13 GPU_NODE_LIST = 'gpuNodeList'
14
15 # function that encrypts the node name passed as argument
16 def encryptNodeName(nameArg):
17     # [Redacted encryption code]
18
19 # Function opens a file using the argument as filename, reads
       the contents into a list and returns it.
20 # Function includes logic to enable encryption of node names
     by reading the numbers from the nodenames and encrypting
     them separately
21 def loadNodeList(fileName):
22     nodes = []
23     with open(fileName) as f:
24         for line in f:
25             toAppend = line.strip('\n').strip(' ')
26             # to enable on read encryption uncomment this
     line
27             #toAppend = encryptNodeName(toAppend)
28             nodes.append(toAppend)
29     return nodes
30
31 # deprecated function that is no longer in use
32 def loadFile(fileName):
33     counter = 0
34     d = {}
35     with open(fileName) as f:
```

```
36          for line in f:
37              d[counter] = line.strip('\n')
38              counter += 1
39      return d
40
41  # the string to num functions attempt to convert the argument
        to float or int respectively, if they fail the print an
       error message and then return
42  # these are being used to validate the inputs from compute
       nodes and gpu nodes respectively, if gpu is true then it
       converts to int these should be combined into one function
43  # once it has been normalized what 'no reading' looks like
       from nodes
44  def stringToNum(string, gpu):
45      if(gpu):
46          try:
47              return int(string)
48          except ValueError:
49              print("Value error int on " + string)
50      else:
51          try:
52              return float(string)
53          except ValueError:
54              print("Value error float on " + string)
55
56      print("conversion error")
57
58  # function checks the if any indexes are missing and injects
       a no measurement value if that index does not exist
59  # if it does exist it makes sure that the value given is
       converted to a numeric instead of storing it as a string
60  # in case of gpu's it appends 0 else -1.0, however this needs
        to be normalized such as no measurement should always be
       represented the same way.
61  def convertAndPad(values, indexes, gpu):
62      valueList = list()
63      if (len(values) != len(indexes)):
64          test = dict(values) #converting this to dictionary
       for easy lookups
65          # this for loop specifically goes through the list of
        indexes, if that index does not exist in the dataset
66          # then inserts a non measurement value into the
       metric, non measurement value needs to be normalized to a
       specific value
67          for index in indexes:
```

```
68              if(index not in test):
69                  if(gpu):
70                      valueList.append(0)
71                  else:
72                      valueList.append(-1.0)
73              else:
74                  valueList.append(stringToNum(test[index], gpu
    ))
75      else:
76          # note value in this case is a tuple (time, value) so
     when appending it has to be indexed in this way value[1]
77          # note that this is different from the above code due
     to that code using a dictionary where the time has become
     the index
78          # potentially could use dictionary for both parts of
    this loop to simplify the function.
79          for value in values:
80              valueList.append(stringToNum(value[1], gpu))
81
82      return valueList
83
84
85 '''
86 Structure of json file
87 data
88      result[ #result is an array of subtrees each of which
    contains the metric and values branch
89          metric[
90              instance # this stores the full node name r##n##.
    lisa.surfsara.nl
91              minor number # this stores the iteration for that
     specific node, e.g. in nvidia metrics '0' is the first
    gpu and '3' is the fourth gpu
92              #few other ones that are not as relevant but may
    be worth exploring
93          ]
94          values[
95              (timestamp, measurement)    #in the array of
    values the data is stored like this.
96          ]
97          .
98          .
99          .
100         .
101     ]
```

```
102 '''
103 # reads the json input and splits it indo nodes and metrics
      respectively, then returns a dictionary containing all the
       nodes and their values respectively.
104 # includes a comment block used for encryption of node names
      during parsing
105 # with the data being a nested datastructure to reach the
      metrics and node information you need to dig into it
106 # TODO::Handle special cpu cases where we are getting results
       per cpu, needs to be designed to handle that somehow
107 def splitJson(jsonObject, dict, indexes, queryName):
108     dict.clear()
109     for result in jsonObject['data']['result']:
110         if 'instance' in result['metric']: #check if result
      contains no metric
111             k = result['metric']['instance']
112             v = result['values']
113 
114             # will skip all non compute nodes, this means no
      administration node, software node, login node or
      fileserver is included in the metrics
115             if not (k.startswith('r')):
116                 continue
117 
118             # node names come as a full address on lisa to
      make the dataset easier to read we prune the node down to
      just r##n##
119             k = k.split('.', 1)[0]
120 
121             # to enable on parse encryption of node names
      uncomment this line
122             # k = encryptNodeName(k)
123 
124             valueTupleList = [i for i in v]
125             values = convertAndPad(valueTupleList, indexes,
      False) # because there is no gpu present in this split
      this passes false
126             dict[k] = values
127 
128 # same intro as splitJson, however this code is only valid
      for metrics in nvidia so the potential to merge the
      functions and handle nvidia using helper functions
129 # may be a much better solution.
130 # difference is this function handles nvidia metrics
      specifically because they come in 4 results for each node.
```

```
131 # it compares the currently read node to the previously read
    node, once there is a mismatch it calls packToInt64 which
    returns a list that is added
132 # to the dictionary, instead of reading node names this
    function could be modified to read minor number and watch
    for it to reset to 0,
133 # so every time minor number hits zero it would package the
    values.
134 def splitGPU(jsonObject, dict, indexes, queryName):
135     prevNode = 'NONE'
136     prePack = []
137     minorNum = 0
138 
139     dict.clear()
140     for result in jsonObject['data']['result']:
141         if 'instance' in result['metric']: #check if result
    contains no metric
142             k = result['metric']['instance'].split('.', 1)[0]
143             v = result['values']
144 
145             if not (k.startswith('r')):
146                 continue
147 
148             if(prevNode == 'NONE'):
149                 prevNode = k
150 
151             # if the previous node is not equal to the
    current node then the integer packaging needs to be done
    and the lists cleared
152             # could also simplify this by just checking the
    minor number and if the minor number in the newly read
    tree is = 0 then
153             # you have to package the values before loading
    in the new node.
154             elif(prevNode != k):
155                 dict[k] = packToInt64(prePack)
156                 prePack = []
157                 prevNode = k
158 
159             minorNum += 1
160 
161             valueTupleList = [i for i in v]
162             values = convertAndPad(valueTupleList, indexes,
    True)
163 
```

```
164             # if nvidia gpu power is found then the queryName
     should be modified to
165             # be nvidia_gpu_power_watts so that we can claim
    correctness in naming since at the moment
166             # it is milliwatts while the data represented is
    in watts due to this modification
167             # this modification and the memory one are here
    to make sure the data fits into 64 bit values when
    packaged
168             # another way would be to modify the queries to
    prometheus to include the calculations to do this
    calculation so that it is unnecessary here
169             #modified the branches to simplify the code
170             if (queryName.startswith('nvidia_gpu_power')):
171                 values = [int(x / 1000) for x in values]
172                 # newValues = [int(x / 1000) for x in values]
173                 # prePack.append(newValues)
174 
175             elif (queryName.startswith('nvidia_gpu_memory')):
176                 values = [int(x / 1048576) for x in values]
177                 # newValues = [int(x / 1048576) for x in
    values]
178                 # prePack.append(newValues)
179 
180             # if neither of the previous ones is true then it
     the function just appends the value unchanged
181             prePack.append(values)
182 
183     # checks if prepack contains information from one final
    node, if there isn't it returns else it
184     # runs the pack to int once more
185     if(len(prePack) == 0):
186         return
187 
188     dict[k] = packToInt64(prePack)
189     prePack = []
190 
191 # packs the values from a list of lists into a 64 bit value
192 # expects this format prePackaged[[value],[value],[value],[
    value]] where each value is a 16bit number or less
193 # this function then uses bitwise or '|' and shifting '<<' to
     pack them into a 64 bit value
194 # and creates a new list called postPack[] that consists of
    all the lists inside prePackaged combined into one
195 def packToInt64(prePackaged):
```

```
196     postPack = []
197     for n in range(len(prePackaged[0])):
198         c = 0
199         for j in range(len(prePackaged)):
200             if(j > 3):
201                 break
202             c = c | (prePackaged[j][n] << 16 * j)
203         postPack.append(c)
204 
205     ''' Consider
206         if(max(postPack).bit_length() > 64):
207             print warnings
208     '''
209     if(max(postPack).bit_length() > 64):
210         print('Max value larger than 64 bits')
211         print(max(postPack))
212         print('len of j: ' + str(len(prePackaged)))
213     #if(max(postPack) > 0xFFFFFFFFFFFFFFFF):
214 
215     return postPack
216 
217 # opening function to the script, takes as an argument the
      jsonObject containing the query result
218 # the name of the query and the list of indexes for the day
      that is being parsed
219 # the index list will have to be generated elsewhere, it's
      from midnight to midnigt - 15sec, with 15 second intervals
       so 5760 values.
220 def pandasParq(jsonObject, queryName, indexes):
221     dict = {}
222     #the reason for loading in the nodelists for gpu nodes
      and then all nodes
223     #is to validate if any nodes are missing in the dataset,
      however since nodelists in the future will not be static
      this requires redesign
224     #one issue with parquet is that if there is a difference
      between two files e.g. column wise etc then you have to
      merge the parquet datasets
225     #however when trying to merge such large datasets i ran
      into issues with memory, finding a reliable source of
      nodelists outside of having to manually update
226     #would be an amazing solution
227     nodeList = loadNodeList(NODE_LIST)
228     nodeList.sort()
229     gpuNodeList = loadNodeList(GPU_NODE_LIST)
```

```
230     gpuNodeList.sort()
231     gpu = False
232 
233     if(queryName.startswith('nvidia')):
234         gpu = True
235 
236     #different split functions if you have gpu or non cpu
    metric, would like to merge those and create sub functions
     that they call with.
237     if(gpu):
238         splitGPU(jsonObject, dict, indexes, queryName)
239     else:
240         splitJson(jsonObject, dict, indexes, queryName)
241 
242     # validation code - might be best to separate this into a
     separate function
243     # this checks the dict against the nodelist
244     # if the node is not in the dictionary it calls
    convertAndPad with an empty list, indexes and true/false
    depending on what is needed.
245     # convertAndPad then generates a list filled with 0 or
    -1.0 depending on gpu or not, becomes simpler if we make
    sure all non readings are the same
246     if(gpu):
247         for node in gpuNodeList:
248             if not node in dict and len(dict) > 0:
249                 values = convertAndPad([], indexes, True)
250                 dict[node] = values
251     else:
252         for node in nodeList:
253             if not node in dict and len(dict) > 0:
254                 values = convertAndPad([], indexes, False)
255                 dict[node] = values
256 
257     # ordering the columns of the dictionary after validation
     and adding missing nodes ensures that all parquet files
    are the same.
258     # note this does not reorder the values in the dataset
    only ensures that the columns are always in the same order
    .
259     oDict = OrderedDict(sorted(dict.items()))
260 
261     #checks if the program indeed got results from the
    provided metric. this is a validation check to make sure
    data was
```

```
262     #returned from this query
263     if indexes and len(dict) > 0:
264         #maybe can use the nodelist when creating the
    dataframes instead of the dicts, then loop over the dict
    and set the values to the
265       #correct frame
266         df = pd.DataFrame(data=oDict, index=indexes, dtype=
    object, copy=False)
267         writeParquet(df, queryName)
268 
269 #saves to parquet dataset from the dataframe and using the
    queryName for the folder to save it to
270 def writeParquet(df, queryName):
271     df.index.name = 'time'
272     df = df.astype('Int64')
273 
274     table = pa.Table.from_pandas(df, preserve_index=True)
275 
276     #the write to dataset will generate the parquet file as
    part of the dataset in the existing folder, in case of no
    dataset being located
277     #at the root_path it write_to_dataset will start it
278     #this could be worth a redesign to open the existing
    dataset if it is there and appending the new data to the
    dataset
279     #TODO::adjust queryname for gpu memory since it is
    modified to be Megabytes and gpu power since it is watts
    not milliwatts
280     pq.write_to_dataset(table, root_path="/project/kristian/
    lisa.parquet/" + queryName, flavor="spark")
```