HPC Job-Monitoring with SLURM, Prometheus and Grafana

Bachelor Thesis

University of Basel
Faculty of Science
Department of Mathematics and Computer Science
HPC Research Group

Examiner: Prof. Dr. Florina Ciorba
Supervisor: Thomas Jakobsche

Author: Pascal Kunz
Matriculation Number: 2018-058-529
pasca.kunz@stud.unibas.ch

May 14, 2022

Acknowledgements

First of all, I would like to thank Prof. Dr. Florina Ciorba for giving me the opportunity to do my bachelor thesis in her group on such an interesting topic. Also, I am especially grateful for the advice and insights provided by my supervisor Thomas Jakobsche during the past three months. Apart from that, I wish to thank Robert Frank, the system administrator of the miniHPC, for his help in deploying the developed software. Lastly, I want to express my gratitude for my family for helping me throughout my journey.

Abstract

An important step in increasing the observability and efficient usage of computational power on High Performance Computers is to capture data about running jobs, consequently storing it and creating means of appropriate visualizations. Users that schedule jobs on HPC systems rely on workload managers that distribute the work across computing units in order to achieve efficient system usage. Another interesting property of workload managers is that they provide a wide array of metrics about running jobs that open the door for the interpretation of anomalous behaviour of jobs such as load-imbalance, CPU- and I/O-anomalies or memory leaks. In this work, the current monitoring of the monitoring of the miniHPC cluster operated by the University of Basel is extended by a new monitoring framework to enable job-level monitoring. We use data provided by SLURM, the miniHPC's workload manager, store it in a memory efficient way using Prometheus, and subsequently visualize it using the interactive web-based application Grafana. Additionally, we simulate a variety of anomalous jobs and evaluate the capability of the monitoring framework to automatically detect abnormal behaviour of jobs by creating appropriate visualizations. The conducted experiments show that our monitoring set-up is able to detect and flag artificially generated anomalies such as load-imbalance, memory leaks, I/O- anomalies as well as CPU-anomalies. Additionally, the framework serves as proof of concept for the capabilities of the combination of SLURM, Prometheus and Grafana for job-level monitoring.

Contents

Chapter 1

Introduction

HPC systems are indispensable constituents for science with various applications in the calculation of partial differential equations, graph problems or stochastic systems [50, 42, 43] that require huge amounts of computing power. As performance and reliability of HPC systems are essential quality features, the monitoring of relevant hardware-interfaces is crucial. Apart from the deriving conlusions about the HPC system's state solely from a systems level, metrics can also be collected on a job-level. As submitted jobs are the main cause of computational work, it is important to gain more insights as to how processes are being distributed in the HPC system and how they interact with the hardware.

1.1 Motivation

Users that execute applications on HPC systems rely on a workload manager that decides how work is distributed across the computing nodes to achieve a high rate of parallelism and the lowest possible execution time. Workload managers such as SLURM [45] define an array of useful user-commands with which important data of running jobs can be queried which can help in the detection of job-anomalies such as load-imbalance, memory-leaks, CPU-variations or I/O-bandwidth issues. As jobs are, abstractly speaking, a cause of computational work that do not reveal their true nature, which is the source code, computational facilities can be misused by jobs that violate the purpose of the system (e.g. bitcoin miners on HPC systems [44]). The memory efficient storage of job-data can be a contributing factor for developing predictive models that allow for the automated detection and termination of such processes. Additionally the storage of data in a suitable database plays a big factor for being able to use state-of-the-art interactive visualization tool-kits such as Grafana [8] that allow for creating concise dashboards that help in the interpretation of job-data and potential anomalies at run-time.

1.2 Current monitoring and limitations

The University of Basel's HPC Group maintains a HPC (called miniHPC) for research purposes [16]. Currently, there exists a monitoring web-based application [17] that provides information about the system's usage. The web-interface mainly consists of two features:

- Display information about the current state of the nodes of the miniHPC. For each node, there is a list providing information about the activity, CPU usage, usage of disk and network interfaces,
- Display long term statistics of various metrics like CPU, Disk, Load, Memory, Network, Processes for a specific node.

However, the current monitoring system has certain limitations, namely that the user interface looks outdated due to its typesetting and color scheme. The plots that can be displayed are pre-configured by the system which implies that the user is not able to create new plots for instance by using database queries. Most importantly however, it is not possible to visualize job related metrics in real-time by which system efficiency can be increased potentially.

1.3 Goal and Research Questions

The University of Basel's HPC group is interested in gaining insights about job specific metrics and how jobs use the systems resources in order to increase efficient usage of the miniHPC. The goals of this thesis are to install and configure Prometheus [32], a monitoring tool for time-series data, and Grafana [8], a web-based visualization application, so that the miniHPC can be monitored. Apart from that, software needs to be developed that collects and cleans SLURM job data and ingests it into the time-series database of Prometheus. By defining new PromQL [32] database metrics, the SLURM job data must be made available in Prometheus so that it can be visualized in Grafana. Furthermore, anomalous jobs from an anomaly suite [36] need to be simulated and their behaviour studied with the goal to be able to capture them with Prometheus and Grafana. In a last phase, we will also evaluate the benefits of the Prometheus-Grafana setup and the visualizations produced.

- How can we collect SLURM data and store it in a time-series database?
- What properties do synthetically generated anomalies posses?
- Can we detect synthetically generated anomalies with our monitoring framework?
- What visualization techniques are suited for tagging anomalous jobs?
- Does the monitoring affect performance?
- What advantages and limitations does the monitoring framework possess?

1.4 Outline

This document will firstly introduce the relevant background information necessary to follow this thesis (chapter 2). Consequently, related work will be discussed in chapter 3. In a next step (chapter 4), we will outline the methodology used which includes the explanation how SLURM-data has been collected, cleaned and forwarded to Prometheus. We will also describe how Prometheus has been used for the retrieval of data and for the flagging of anomalous processes. The methodology further deals with what visualization techniques which have proven to be effective for visualizing job data and their anomalies, and also how anomalies have been simulated. Next, we will share the result of our work (chapter 5), and discuss and conclude our work in chapters 6 and 7.

Chapter 2

Background

This chapter introduces concepts and necessary information to follow this thesis.

2.1 Operational Data Analytics

Operational Data Analytics (ODA) is a relatively young term introduced by Bourassa et al. in 2017/2018 [37]. Researchers of the Energy Efficient HPC Working Group (EE HPC [5]) consulted leading HPC facilities worldwide in order to gain knowledge as to how data of their HPC-systems is used with the goal to achieve more energy efficiency. The workflow of data usage, storage and visualization was declared as the process of ODA. MODA, as a related discipline, is also concerned with monitoring processes, meaning how metrics are collected from HPC-system hardware interfaces [44]. It was only back in 2020 when the "moda20: First International Workshop on Monitoring and Operational Data Analytics (MODA)" [18] was held that discussed how HPC facilities use monitoring workflows for their infrastructure, which shows how young of a discipline MODA is in fact. The broader goal of MODA is to find means of monitoring and visualization workflows in order to increase the efficient usage of expensive computational power.

2.2 miniHPC

The University of Basel owns a small but capable HPC system (called miniHPC [16]) that can be used by students or researchers for homework or research reasons.
The miniHPC uses a very common architecture which is called a cluster [51, p. 84]. In fact, as of November 2021, the cluster architecture is prevalent in approximately 92 percent of the top 500 supercomputers in the world [34]. A HPC cluster is a network of interconnected nodes where the nodes are independently functioning computing units consisting of several processors and memory themselves. These nodes are then connected through a layer of network switches. The miniHPC has 5 Network switches that are arranged in a so called fat-tree topology. A fat-tree topology implies that the network switches are arranged as a binary tree and that higher levels of network switches have a higher bandwidth. The lowest layer aggregates all connection coming from the 30 computing nodes via an Ethernet network with a speed of 10 Gbit/s. The second layer in turn connects the lower layer network switches by an Omni-Path network-switch with 100 Gbit/s

2.3 Resource Management using SLURM

Jobs that are running on the miniHPC need to be distributed across the computing nodes in order to achieve a high rate of parallelism. Workload managers help to achieve this as they are able to allocate processes to computing resources. The miniHPC uses the SLURM [45] workload manager. The key functionality of SLURM as summarized on the official website of SLURM [45] is to grant access to computing resources (nodes) to users for job execution. SLURM comes with an array of user commands that allows for the submission of jobs as well as for the monitoring of running jobs. SLURM additionally contains a conflict resolving mechanism caused by jobs that have demanded the same resources. It can do so as jobs are en-queued and handled regarding their priority thanks to a priority-queue of submitted jobs. The software-architecture of SLURM uses a hierarchical design of daemons that are called slurmctld (controller) and slurmd (Figure 2.1).

The key functionalities of the daemons was outlined in a paper by Jette et al. [28] as part of an official documentation about the systems architecture and is summarized in the following.

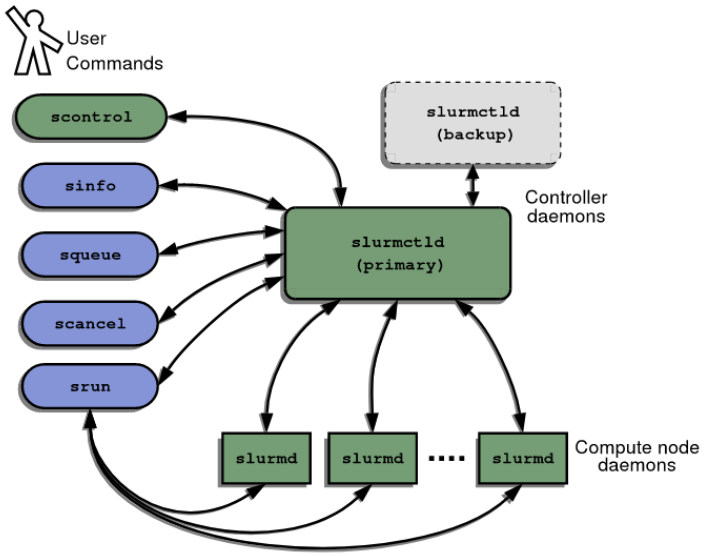

Figure 2.1: SLURM architecture overview taken from [28]

The daemon slurmctld is the central daemon that gives instructions to the lower level slurmd daemons. It is installed on one node of the miniHPC. For one it serves as a Node Manager that instructs the slurmd daemons to give updated metrics about the Nodes. Furthermore, it provides partition management functionalities. Lastly, it serves as a job manager by maintaining a queue of pending jobs and allocating the jobs to computing resources.

The daemon slurmd runs on top of the computing nodes and is used for starting and terminating jobs coming from srun command or the slurmctld daemon. Furthermore it reports the state of the nodes by capturing metrics from well defined interfaces and forwards them to the slurmctld

daemon.

2.4 Job anomalies

Job anomalies can be described as the behaviour of software that can potentially negatively influence the overall efficient usage of HPC-power. Anomalies can present themselves in various parts of the HPC system's hardware while a job is being executed. In the following we will introduce the reader to common anomalies with their causes and how they affect the systems hardware. The examples are based on anomalous applications outlined as part of the HPAS anomaly suite by Ates et al. [36] that are going to be discussed in chapter 4.

Central Processing units (CPU) can be subject to performance variations caused by applications that execute a lot of instructions per second resulting in a high CPU usage and high CPU time. Contrarily, it is possible that instruction execution is put on hold because of inefficient programming or by using programming-functions that put the CPU in idle mode.

Applications can additionally influence memory hardware caused by iteratively allocating memory at run-time (memory leak) or by allocating huge blocks of memory at the start of the program (memory eater). These anomalies often times manifest themselves in increased virtual memory (VM) size or resident set size (RSS). Virtual memory is the technique of operating systems that allocates processes to part of the Random Access Memory (RAM) and hard-disks whereas RSS is the amount of memory allocated only to the RAM. Memory leaks or memory eaters are problematic as they allocate memory not only from Random Access Memory (RAM) but also from the disk space that potentially can cause the termination of other running processes.

As computer programs are able to read and write files into the file system, it is worth observing the amount of bytes read or written by the application during the execution time. Similarly to memory leaks, I/O bandwidth anomaly can be caused by programs that iteratively create files and read their contents due to programming error or malicious intent.

Another important factor for increasing the efficient usage of the computing resources is to consider the amount of load imbalance caused by processes. Load imbalance manifests itself in the fact that the allocated resources do not contribute to solving the problem to the same extent. This poses the problem that nodes that solve their tasks with a delay cause the execution time of the program to be prolonged.

2.5 Time-Series and Time-Series Databases

For the scope of this project it is important to understand the concepts of time-series and the respective storage units for them which can be referred to as time-series databases. Time-series are ordered sets of data-points that contain a measurement value as well as a timestamp. The monitoring of jobs via the workload manager SLURM can generate job metrics on a frequent basis. By allocating the measurement values to the time that they have been collected, time-series data are produced. The Prometheus monitoring tool that is going to be introduced in more detail in the methods (chapter 4) builds its data-model on the concept of time-series and stores it in a time-series

database. Other paradigms of database architecture such as relational databases are not suitable for time series as they are not configured to handle the huge amounts of ingestions possibly coming from the monitoring of devices as they use another data-model and ingestion mechanisms. The compression of data is another very important factor that makes time series data even more usable for huge amounts of data. Prometheus uses two compression algorithms that can compress the 64 bit timestamp and measurement value down to 1.37 bytes [48]. Prometheus uses encryption algorithms proposed by Pelkonen et al. of the Facebook incorporated that was included in the Gorilla time-series database for the compression of time-stamps and measurement values [48]. The following paragraphs briefly introduce the idea of the compression algorithms.

Double Delta Encoding for time-stamps

The idea of the compression algorithm (as outlined in [48]) is instead of storing the full timestamp for every measurement, to store the differences of differences of time between two timesteps. This process is referred to as calculating the delta of deltas. For instance the time values 1471238510, 1471238515, 1471238520, 1471238525 can be compressed in the delta compression as 1471238510, +5, +5, +5. Going even further to the delta-of-delta compression, this implies that this time-series can be compressed as 1471238510, +5, +0, +0.

XOR compression for measurement values

The idea (as outlined in [48]) is instead of storing every measurement value completely, the exclusive or (XOR) operation is applied to two consecutive measurement values. It makes use of the fact that the XOR of consecutive measurement values often produces trailing and leading zeros in floating point representation which can be omitted, which saves memory.

Chapter 3

Related Work

This section provides an overview to the related work for the topic of existing cluster monitoring frameworks. The frameworks will be introduced by summarising the essence of their functionalities and findings.

3.0.1 Ganglia Monitoring

Ganglia is a monitoring framework that makes use of an hierarchical arrangement of daemons which can be seen in Figure 3.1 [46].

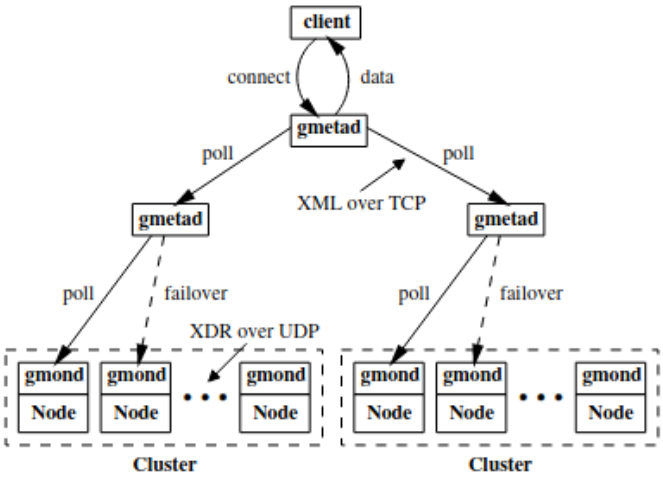

Figure 3.1: Ganglias Architecture [46]

The ganglia monitoring daemon (gmond) runs on top of computing nodes where it collects up to 37 different metrics [46]. Gmond consists of threads for collecting node data and for updating metrics coming from different nodes. The Ganglia Meta Daemon (gmetad) can poll data from gmond and forward it to to its hierarchically superior gmetad entity. The Ganglia monitoring framework

includes a visualization toolkit called RRDTool (Round Robin Database) [27] using which the data can be visualized in a web-based user interface. The University of Basel currently operates the Ganglia framework for monitoring the miniHPC with a web-based graphical user-interface [17]. The capabilities and limitations of the UI have been discussed in chapter 1.2.

3.0.2 MAP: Monitor-Analyzer-Predictor

In 2020 Pal et al. of the Indian Institute of Technology (IIT) Kanpur defined a framework for monitoring analysis and prediction (MAP) for jobs running on HPCs [47]. The framework separates monitoring, analysis and prediction into three separate modules. MAP makes use of the fact that a wide variety of job-data can be captured with job schedulers. Therefore the systems monitoring is based on the PBS job scheduler [41] using which real-time job metrics are parsed into JSON-format so that data can be displayed into a web-based user interface. Though MAP in its initial form was created for PBS job scheduler and especially configured for the qstat command, it is possible to use other different schedulers job statistics outputs with the MAP framework [47]. Apart from that, the monitoring module also captures interesting data about the topology of in-use nodes while a job is running which serves the purpose of gaining a deeper understanding how the job is distributed across the HPCs node network. The analysis module parses job-metrics log-files into a MYSQL Database from which long-term analysis graphs source their information from. Using the job-logs information, MAP is also able to calculate expected wait-times for jobs in pipelines.

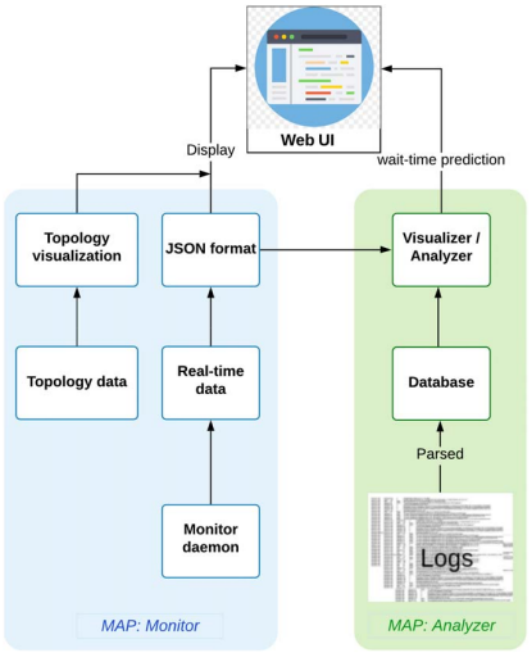

Figure 3.2: MAP architecture [47]

3.0.3 Likwid Monitoring Stack and Cluster Cockpit

In 2017 Röhl et al. proposed the Likwid Monitoring Stack [49]. Similarly to the MAP framework it also relies on capturing job metrics using job schedulers but it can additionally also capture metrics from currently existing monitoring systems. Jobs are tagged with identifiers and forwarded via the HTTP interface to time-series databases for users and system administrators. Likwid uses the Influx time-series Database [13] which features a SQL like query language. For the visualization Grafana [8], an interactive and webbased graphing tool, was used. Two years later, in 2019, the Likwid Monitoring Stack was extended by Eitziner et al. [40] with a monitoring set up called Cluster Cockpit. Cluster Cockpit is similar to Likwid in the monitoring process and storage of the data but it has a distinct and self-programmed data visualization interface. Especially interesting is the feature that lets the user tag jobs with job-tags using the Plotly.js charting library. Furthermore the system is able to automatically tag jobs based on user-defined rules. The graphical user interface is grouped in rows, where one row holds information about specific jobs such cpu-load or used memory by the application.

3.0.4 PIKA Framework

In 2020, Dietrich et al. proposed PIKA [39]. PIKA differentiates the collection, storage, analysis and visualization of data. Data is collected from workload manager daemons that are running on the computing nodes which is then going to be stored in time-series databases; one for the short term data and the other for the long term. Apart from that PIKA collects job metadata which are essentially the execution parameters provided from the batch system which are going to be stored in a relational database. The analysis module consists of tagging jobs into defined categories. The visualization is done by using tables and time series plots in the web application framework called Angular [2].

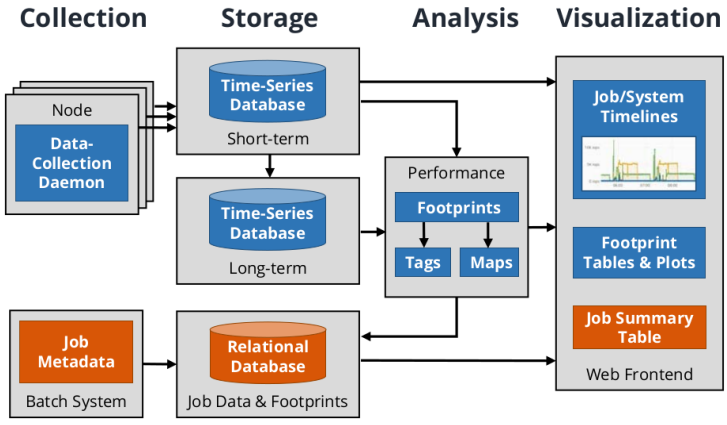

Figure 3.3: PIKA architecture [39]

3.0.5 Our solution approach

Our solution approach for the problem specified in the introduction also uses the concept of capturing data of running jobs using a workload manager, storing the data in a time-series database, labelling anomalous jobs using certain characteristics and then using an interactive visualization tool for creating tables graphs and alerts similarly to the PIKA framework [39]. It therefore also differentiates between collection, storage, analysis and visualization modules that are integrated into a full monitoring framework that can be seen on Figure 3.4. Our approach differs from PIKA in the sense that we do not intend on differentiating between a short- and longterm database and that we do not capture job metadata. We are only going to focus on data coming from running jobs that can be monitored with a workload manager. SLURM that runs on top of the computing nodes is able to capture valuable metrics with which we want to be able to capture memory leaks, CPU contention and anomalies concerning the I/O bandwidth. The Prometheus monitoring tool includes a time-series database and is going to be introduced in the methods section in more detail and that we will use as a storage component. We intend on creating visualizations that automatically tag and visualize jobs that contain certain anomalies using the interactive visualization tool Grafana.

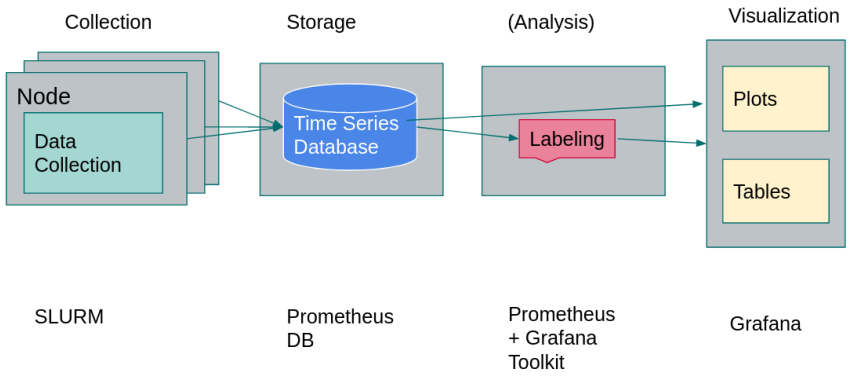

Figure 3.4: Our monitoring framework (subset of the PIKA monitoring framework [39])

Chapter 4

Methods

This chapter outlines the tools used for the monitoring setup.

4.1 Getting the data using SLURM

We have outlined the key functionalities and properties of the SLURM workload manager in the background section above. In the context of our monitoring framework, SLURM can be used in order to collect information about the system and most importantly about running jobs which we want to load in to a suited database.
In SLURM, there exists a wide array of useful user commands for gaining more insight about how processes use the system. Two of them are especially useful, namely the sstat and sacct command [33, 30]. The sstat command can be used in order to get metrics of running jobs whereas the sacct command can retrieve job-data stored in the log-file or SLURM database of jobs that have already terminated. Output is generated in the Linux terminal of the miniHPC by invoking sstat and sacct with configurable flags and so called job status fields which define the metrics for which data will be displayed. In the following list, the reader will find the job status fields used and their descriptions. The names and descriptions of the job status fields are taken from the official documentation of the sstat and sacct command [33, 30].

4.1.1 Used job status fields

Job status field	**Sample**	**Description**
AveCPU	00:02:03	Average (system + user) CPU time of all tasks in job.
AveCPUFreq	22781.13K	Average weighted CPU frequency of all tasks in job, in kHz.
AveDiskRead	11812800	Average number of bytes read by all tasks in job.
AveDiskWrite	1705	Average number of bytes written by all tasks in job.

AvePages	0	Average number of page faults of all tasks in job.
AveRSS	93429606	Average resident set size of all tasks in job.
AveVMSize	3680955468	Average Virtual Memory size of all tasks in job.
ConsumedEnergy	0	Total energy consumed by all tasks in job, in joules. Note: Only in case of exclusive job allocation this value reflects the jobs' real energy consumption.
JobID (sstat)	786103.0	The number of the job or job step. It is in the form: job.jobstep
MaxDiskRead	11820140	Maximum number of bytes read by all tasks in job.
MaxDiskReadNode	cl-node001	The node on which the maxdiskread occurred.
MaxDiskReadTask	0	The task ID where the maxdiskread occurred.
MaxDiskWrite	22256	Maximum number of bytes written by all tasks in job.
MaxDiskWriteNode	cl-node001	The node on which the maxdiskwrite occurred.
MaxDiskWriteTask	0	The task ID where the maxdiskwrite occurred.
MaxPages	0	Maximum number of page faults of all tasks in job.
MaxPagesNode	cl-node001	The node on which the maxpages occurred.
MaxPagesTask	0	The task ID where the maxpages occurred.
MaxRSS	105652224	Maximum resident set size of all tasks in job.
MaxRSSNode	cl-node006	The node on which the maxrss occurred.
MaxRSSTask	100	The task ID where the maxrss occurred.
MaxVMSize	3745497088	Maximum Virtual Memory size of all tasks in job.
MaxVMSizeNode	cl-node006	The node on which the maxvsize occurred.
MaxVMSizeTask	108	The task ID where the maxvsize occurred.
MinCPU	00:02:03	Minimum (system + user) CPU time of all tasks in job.
MinCPUNode	cl-node002	The node on which the mincpu occurred.
MinCPUTask	31	The task ID where the mincpu occurred.
NTasks	160	Total number of tasks in a job or step.
ReqCPUFreq	Unknown	Requested CPU frequency for the step, in kHz.
TresUsageInAve	cpu=00:02:26	Tres average usage in by all tasks in job. NOTE: If corresponding TresUsageInAvTask is -1 the metric is node centric instead of task.
TresUsageInMin	cpu=00:02:26	Tres minimum usage in by all tasks in job. NOTE: If corresponding TresUsageInMinTask is -1 the metric is node centric instead of task.
TresUsageInMax	cpu=00:02:26	Tres maximum usage in by all tasks in job. NOTE: If corresponding TresUsageInMaxTask is -1 the metric is node centric instead of task.
TRESUsageInMaxNode	cpu=00:02:26	Node for which each maximum TRES usage out occurred.
JobID (sacct)	784315	The identification number of the job or job step.
User	fewaki31	The user name of the user who ran the job. .
Jobname	hpl-8	The name of the job or job step.

AllocNodes	8	Number of nodes allocated to the job/step. 0 if the job is pending.

Table 4.1: SLURM sstat and sacct job status fields, job status fields and descriptions taken from [33, 30] descriptions taken from [33, 30]

4.1.2 Data pre-processing

In order to obtain all the relevant job information provided from table 4.1 above, a shell script had to be developed that executes the sstat command with all the relevant job status fields and additionally merges it together with the output from the sacct commands for jobid, user- and jobname.

```
#!/bin/sh
sacct --state=RUNNING --format=JobID,user,jobname,allocnodes -n -P |
while read -r line;
do
substrings=(${line//|/ })
jobid=${substrings[0]}
name=${substrings[1]}
username=${name}
jobname=${substrings[2]}
allocnodes=${substrings[3]}
 if [[ $jobid =~ ^[0-9]+$ ]];
 then echo $username"|"$jobname"|"$allocnodes"|"$(sstat -j "$jobid"
 --format=JobID,avecpu,avecpufreq,
          avediskread,avediskwrite,avepages,
          aveRss,avevmsize,consumedenergy,maxdiskread,
          maxdiskreadnode,maxdiskreadtask,maxdiskwrite,
          maxdiskwritenode,maxdiskwritetask,maxpages,
          maxpagesnode,maxpagestask,maxrss,maxrssnode,
          maxrsstask,maxvmsize,maxvmsizenode,maxvmsizetask,
          mincpu,mincpunode,mincputask,ntask,
          tresusageinav,tresusageinmax,tresusageinmin,
          TRESUsageInMaxNode -p --noconvert -n);
 fi;
 done
```

Listing 4.1: SLURM Bash Script

By executing the bash script (called fullcommand.sh) above (Listing 4.1), slurm collects the metrics from the computing nodes and prints them into the command-line (Figure 4.1). We took advantage of the sacct and sstat commandline flags -n (prints output without header) -p (parses the output delimited by "|") and –noconvert (prevents conversion from original format, meaning that for all outputs the units remain the same).

Using this script made it possible to query for real-time data of running jobs and gave us a lot of data to be interpreted. As a next step, the data received in the linux commandline needed to be

```
$ sh fullcommand.sh
fewaki31|hpl-8|8|791544.0|00:02:26|22984.30K|11816362|
1778|0|102049689|3681700582|0|11842638|cl-node005|
19|28116|cl-node005|0|1|cl-node005|3|108277760|cl-node012|42|
3745529856|cl-node013|60|00:02:26|cl-node005|11|160|
cpu=00:02:26,energy=0,fs/disk=11816362,mem=102049689,pages=0,vmem=3681700582|
cpu=00:02:26,energy=0,fs/disk=11842638,mem=108277760,pages=1,vmem=3745529856|
cpu=00:02:26,energy=0,fs/disk=11809515,mem=96325632,pages=0,vmem=3676282880|
cpu=cl-node012,energy=cl-node005,fs/disk=cl-node005,mem=cl-node012,
pages=cl-node005,vmem=cl-node013|
```

Figure 4.1: Command-line output provided from SLURM

cleaned, which implied converting the time-values into seconds and omitting unit specifiers (e.g. (00:00:59, 14956.31K, cl-node001) $\mapsto$ (59, 14956.31, 1)). The data-cleaning has been achieved using a GO-language program that can be found in on the github repository of this thesis.

4.2 Prometheus

Prometheus is an open-source framework for monitoring and alerting purposes [32]. It contains a time-series database that comes with its own query language called PromQL[32]. For our monitoring framework, we use Prometheus' database for the storage of the job-metrics and its powerful query language in order to retrieve data.

The general architecture of Prometheus can be abstracted as follows (Figure 4.2):

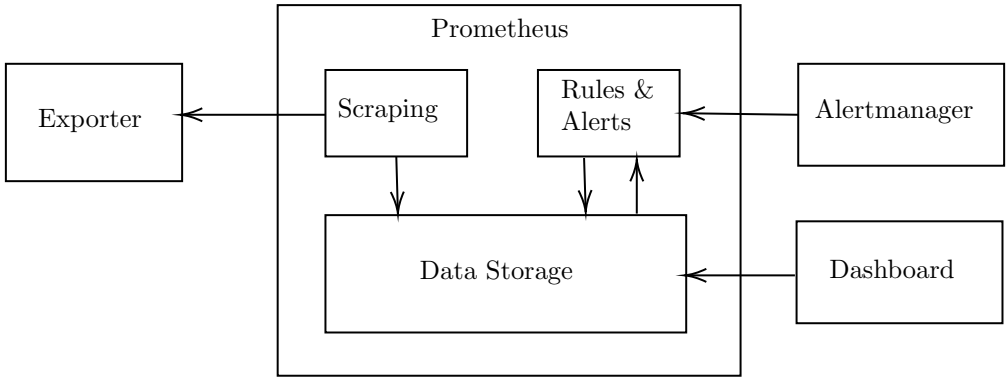

Figure 4.2: Prometheus architecture overview taken from [38, p. 11]

4.2.1 Exporter

Exporters (as outlined in [20]) are used in order to collect and forward metrics to Prometheus from a computing interface via the Hyper Text Transport Protocol (HTTP). Exporters can be developed by any programming language that can achieve forwarding metrics in a specific predefined format to a web-interface (Figure 4.3). However, for the programming languages Java, Scala, Python, Ruby, Rust and Go there exists Prometheus client libraries that help in the collection of data, in the definition of queries and in the forwarding from the monitoring interface to the HTTP interface [4]. We decided to settle with the Go Language that follows the paradigms of a imperative programming language and is also strongly related to C [7]. Our decision was based on the fact that there already exist an open-source exporter called prometheus-slurm-exporter [23]. Although the exporter includes a wide variety of useful cluster specific metrics, it was not capturing information about running jobs. Therefore code had to added in order to capture additional useful metrics.
The code that had to be implemented was used to do the following: For one, it executes the SLURM bash-script (Listing 4.1) and parses the output into 64-bit floats for metrics values and strings for query labels. Additionally, new database queries had to be defined on a code level, using which data would be made available for the user. Furthermore the data had to be transported to an HTTP interface so that the Prometheus client can access the metrics. The full code as well as the added Prometheus Database queries can be found in the repository of this thesis.

4.2.2 Scraping

Scraping (as outlined in [22]) is the process of getting the metrics into the Prometheus database. Prometheus does this by using so called scrapes which are essentially HTTP requests. The scraping frequency is dependent on the settings of the Prometheus' configuration file that is written in the YAML language and which can be found in the appendix (chapter 8) of this document. For the scope of this project, we decided to go for a scraping frequency of 5 seconds that can of course easily be increased or decreased if needed. Chapter 6 discusses the possible performance degradation of the SLURM slurmctld daemon, caused by sending scrapes too frequently. The process of scraping has been abstracted in figure 4.3 below.

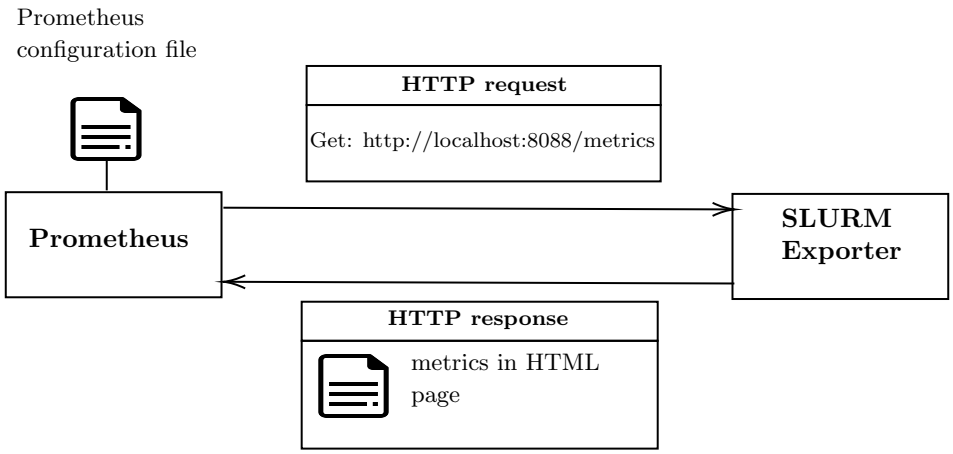

Figure 4.3: Abstracted functionality of scraping

4.2.3 Storage

Prometheus (as outlined in [25]) has its own custom time series database that is stored locally on the hosts computer. Data is stored in folders (Figure 4.4) that contain data for two-hour intervals. Every folder contains chunks (wherein all the actual time-series data is stored), metadata (containing metadata such as number of samples, compression properties and chunk ids) as well as an index file that maps metric names and metric labels to the time-series data.
Per default Prometheus uses the so called double-delta encoding for time stamps that has been and the floating point compression for measurement values that has been introduced in the background section above (Chapter 2) [26, 48].

4.2.4 Datamodel and Querying utilities

Time-series are ordered data points that contain a measurement value and a time-stamp. In Prometheus' case the measurement values and times-tamps are stored exclusively as 64-bit floats[32]. Figure 3.2 specifies the data-model of one Prometheus sample consisting of metrics name, labels, timestamps and the measurement value.

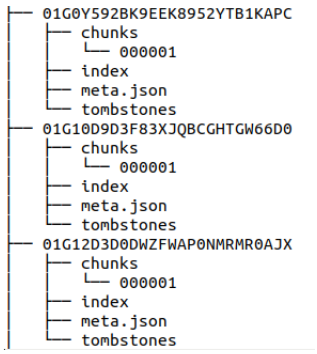

Figure 4.4: Prometheus storage structure

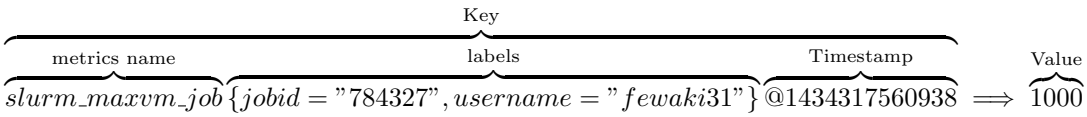

Figure 4.5: Prometheus Data Model [26]

Contrarily to more common databases such as SQL, Prometheus has a unique query language called PromQL that is highly adapted for time-series retrieval. The query language uses the unique characteristics of Prometheus' data-model in order to build queries [24]. In PromQL data is stored as a key-/ value pair where the keys consists of labels and the value being the actual measurement value. PromQL is able to filter query-sets based on labels. Furthermore, query-sets can be combined using binary operators which further extends the monitoring capability. Additionally there exists a wide variety of functions that can be applied to the query-set in order to detect anomalies. For instance, we have applied the deriv() [21] function, that calculates the derivative of a time-series with respect to time in order to flag memory-leaks or I/O-bandwidth anomalies which oftentimes typically produce monotonically increasing functions. In order to detect CPU anomalies, we have applied the function stddev_over_time() [21] which calculates the standard deviation for instance for the CPU-frequency of a running job. For detecting load-imbalance we have applied the sum_over_time [21] for summing up the time-difference of allocated CPU-Time.

4.2.5 Our datamodel and added queries

We have outlined the abstract functionality of exporters above. For the scope of this project this implied that we had to adapt our code to create queries that fit the data model of Prometheus. We have created an array of queries using which all the metrics specified in table 4.1 are covered. Additionally, we have added the labels jobid and username and jobname to the metrics, which makes it easier to group, filter and aggregate prometheus-metrics as part of typical data analysis processes.

Definition of new prometheus metrics in Go

In the following, we will describe how new PromQL database-metrics can be defined for the example of the "slurm_maxrss_runningjob" that returns the maximal resident set size of a job. The github repository [23] can serve as a blueprint for adding new prometheus metrics in Go. The Go-Prometheus library provides the programmer with two useful functions: prometheus.Desc and prometheus.NewDesc. Prometheus.Desc is a struct consisting of values for the name, the help text (that the Prometheus user can use to get more information about a specific function) and the labels of the query. The function prometheus.NewDesc is the generator function of prometheus.Desc. By defining a struct such as JobsRCollector that stored the description entities, the process of creating new metrics is facilitated.

```
 type JobsRCollector struct {
        maxrss:  *prometheus.Desc
}

 func defineMaxRSSMetric() *JobsRCollector {
        labels := []string{"jobs", "username", "jobname"}
        return &JobsRCollector{maxrss: prometheus.NewDesc(
        "slurm_maxrss_runningjob", "maxrss for running job", labels, nil)
        }
}
```

Listing 4.2: Definition of Prometheus Query in Go

In a next step, the actual metrics values needs to be allocated to the prometheus-metric. This can be achieved by using the function "MustNewConstMetric" as follows.

```
 func (jc *JobsRCollector) Collect(ch chan<- prometheus.Metric) {
    maxRssValues, jobid, username, jobname := parseMaxRss()
        for i := range maxRssValues {
                ch <- prometheus.MustNewConstMetric(
                jc.maxrss, prometheus.GaugeValue,
                maxRssValues[i], jobid[i], username[i], jobname[i])
        }
}
```

Listing 4.3: Allocate metrics values to metric

In a last step, the the collector needs to be registered.

```
func init() {
        prometheus.MustRegister(JobsRCollector())
}
```

Listing 4.4: Registration of Prometheus metric

4.3 Grafana Visualization Tool

Though the Prometheus monitoring framework does contain a graphical user interface for entering queries and visualizing data, it is lacking the power, portability and configurability of the Grafana application [8]. Grafana is an interactive data visualization tool that can be characterized by its support of various data-sources and its very user-friendly and intuitive user-interface. Grafana relies on community-driven development with more than 1550 contributers and over 10 million global users with over 800′000 instances being installed on hosting computers worldwide[1].

Grafana has several features and contributes to a lot of use cases. Not only is it suitable for beautiful interactive data-visualizations dashboards but also for simple data exploration. Additionally it contains alert-management utilities, that can notify system administrators and researchers on user-defined alert-cases. Grafana consists out of a wide array of visualization options that cover most data-scientists needs. The options range from simple time-series charts, histograms, heatmaps and bar charts to more sophisticated visualizations such as node graphs, and tables.

Our decision to use Grafana as a visualization toolkit was rooted in the fact that the software is very intuitive to use and is especially well suited for the visualization of time-series data and the Prometheus database. Especially interesting is the fact, that Grafana is steadily increasing in functionality as new plug-ins for even more visualization are being released at a steady pace. This is due to the fact that any user can develop their own dashboards and visualization plug-ins and share it with the community.

After installation, Grafana's client is reachable on port 3000 on the hosts web-browser per default. Prometheus queries that are entered in Grafana's user interface are forwarded to the prometheus client whereupon prometheus returns a JSON response with the relevant data that is going to be visualized in the web-browser (Figure 4.6).

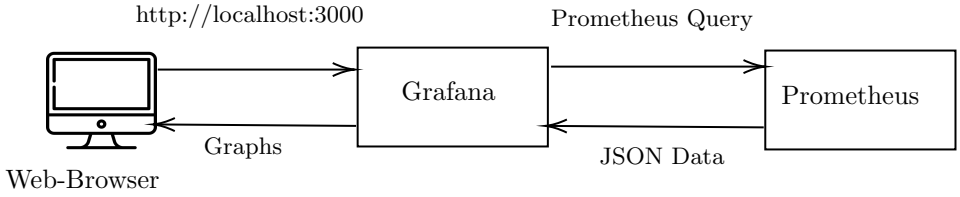

Figure 4.6: Grafana's operating logic

4.3.1 Visualization techniques used

Alertlists

Another interesting visualization tool, especially from a system administrators point of view, are alertlists. Alertlists can be used in order to visualize jobs that have triggered a specific alert-case. Therefore they provide a concise overview of how many alerts are currently being detected by the system and can lead to an improvement of system usage efficiency. The user is able to specify the conditions based on which alerts shall be triggered. The conditions are based on a query and

an expression. Table 4.2 contains the alert-conditions and expressions used in order to trigger the alerts. The defined alert-conditions are the result of our experimentation documented in 5.1.

Alert Name	Prometheus Query (PQ)	Prometheus Expression
High CPU frequency variation	stddev_over_time(slurm_avecpufreq_runningjob[30s]) >= bool 2000	PQ == 1
Low CPU Time	deriv(slurm_avecpu_runningjob[30s]) <= bool 0.95	PQ == 1
Strictly increasing diskwrite	deriv(slurm_maxdiskwrite_runningjob[30s]) >= bool 300	PQ == 1
Strictly increasing diskread	deriv(slurm_maxdiskwrite_runningjob[30s]) >= bool 300	PQ == 1
Strictly increasing VMSize	deriv(slurm_maxvmsize_runningjob[30s]) >= bool 300	PQ == 1
Strictly increasing RSS	deriv(slurm_maxrss_runningjob[30s]) >= bool 300	PQ == 1
Load-imbalance in CPU Time	sum_over_time((slurm_tresusageinmax_runningjob-slurm_tresusageinav_runningjob)[1d:30s]) >= bool 3	PQ == 1
load-imbalance node usage	changes(slurm_tresusageinmaxnode_runningjob[45s]) == bool 0	PQ == 1

Table 4.2: Alert Cases and Prometheus Queries used

Grafana also features the utility to email specific people on these alert cases, which makes the monitoring even more adaptive to the needs of system administrators.

Tables

We have exploited Grafana's table visualization utility [12] in order to come up with a concise overview that displays the currently running jobs together with anomalies. Our decision to pursue this approach was rooted in the fact that tables are extremely ambivalent in their capabilities. For instance, users can adjust column sizes, and alignments and order of columns which allows for a concise overview and of running processes. Additionally, the background color of certain cells can be coloured with contributes to the visual tagging of jobs that contain anomalies, which we were aiming for. Columns have been added to display the following data: *Job-ID, Job-name, Username, CPU-Frequency, CPU-Time,* ***CPU-Anomaly****, MaxVMSize, MaxRSS,* ***Memory-Anomaly****, Maxdiskwrite, Maxdiskread,* ***I/O-Anomaly****, load-imbalance seconds,* ***load-imbalance***
The columns for the anomalies change their color should an alert be triggered.
Of course at some point, the table gets too wide in column size which makes it impractical to add a new column for every collected metric. However, the columns can easily be adapted by the user with regards to size and amount of columns. The results section contains the job table that has been created as a reference (Figure 5.15).

Bar gauges

Grafana bar gauges [9] can be used in order to visually display metrics as a bar. We have used this technique for various visualizations. As an example, we have used bar gauges in order to show the accumulated CPU Time per user over the duration of one day which gives us an indication as to who is using the system to what extent. This could be achieved with query in listing 4.5. Apart from that, bar gauges have also been used in order to evaluate which user triggers the most alert cases. This was achieved by counting the amount of times that specific alert-cases, as outlined in 4.3.1, have been triggered. This was queried with the query in listing 4.6 (Figures 5.17, 5.19 in results).

```
sum(
   sum_over_time(slurm_avecpu_runningjob[1d])
   ) by (username)
```

Listing 4.5: Prometheus Query for summing up the CPU usage over time by user

```
sum(
    count_over_time(
     (
     (deriv(slurm_avecpu_runningjob[30s]) < bool 0.95 or
     stddev_over_time(slurm_avecpufreq_runningjob[30s]) >= bool 3000 or
     deriv(slurm_maxdiskwrite_runningjob[30s]) >= bool 300 or
     deriv(slurm_maxdiskwrite_runningjob[30s]) >= bool 300 or
     deriv(slurm_maxvmsize_runningjob[30s]) >= bool 300 or
     sum_over_time(slurm_tresusageinmax_runningjobslurm-
     tresusageinav_runningjob)[1d:30s]) >= bool 3
     ) == 1
     )[1d:4s])
    )
   by (username)
```

Listing 4.6: Prometheus Query for counting the amount of anomalies caused by users

Status history

Another visualization feature we used was Grafana's status history [11]. This is an interesting visualization technique as it can provide valuable insights as to at what points in execution-time a job produces anomalies. JobIDs and their corresponding username are listed on the y-axis. The actual plot is distributed across x-axis as a horizontal bar. For each point in time that a job has triggered an alert, the the status history will be colored in red whereas normal status will be colored in green (Figure 5.16 in the results as reference).

4.4 Software for simulating anomalies

In order to simulate *load-imbalance*, we have used a C program (called mandel.c) that computes the Mandelbrot set. The program was run with the Open Multi-Processing API [19] and a static scheduling technique and is a load-imbalanced application.

4.4.1 HPAS: Performance Anomaly Suite

The HPC Performance Anomaly Suite (HPAS) invented by Ates et al. [36] is a module for producing HPC-jobs with synthetic anomalies which allows researchers to study anomalous behaviour of job execution. The anomalies produced with HPAS targets CPUs, cache, memory, the clusters network of nodes as well as filesystems. For the scope of our project we have decided to use HPAS in order to simulate anomalies and study their behaviour. Our decision was based on the fact that in order to be able to flag anomalies, we would first have to understand their behaviour and evaluate what SLURM metrics are suitable in order to capture them.

HPAS' codebase [6] contains 8 anomalies, written as C and shell code, that are going to be summarized in the following. The information has been gathered from [36] and [6].

Application Name	Description
cachecopy.c	Read and write in cache
cpuoccupy.c	Set CPU usage to defined percentage
iobandwidth.sh	Read and write into files
iometadata.c	Create and delete files
membw.c	Stack memory write
memeater.c	allocate, fill and release huge blocks of memory
memleak.c	Iteratively allocate and fill memory
netoccupy.c	send message among two nodes

Table 4.3: HPAS applications and their behaviour [36, 6]

cachecopy.c

cachecopy.c as uses the cache intensively. The c code produces two arrays that continuously copy their contents from one array to the other. The two arrays are allocated to either the L1, L2 or L3 cache depending on what the user chooses. The chachecopy.c anomaly can therefore cause cache memory lines to be overwritten frequently. [36, 6]

cpuoccupy.c

cpuoccupy.c is a program that can execute code with a user-defined utilization of the CPU. The code continuously executes arithmetic operations and execution is put on hold a specified amount of time, depending on the utilization percentage specified by the user, causing the CPU usage to drop significantly. [36, 6]

iobandwidth.sh

iobandwidth.sh is used in order to increase the I/O bandwith significantly. It does this by iteratively generating directories in a while loop and moving artificially generated input files into them and reads their inputs. The user is able to define the sizes of the files created. [36, 6]

iometadata.c

iometadata.c iteratively creates files, writes a random character into it and then closes and deletes them again after 10 iterations. The user is able to define the rate at which files are created. [36, 6]

membw.c

membw.c targets the stack-memory of the running application by allocating two arrays in the stack. The code continuously rewrites the contents of the stack's memory of both arrays during the execution time of the anomaly. [36, 6]

memeater.c

memoryeater.c allocates a user-defined amount of memory to the application at the beginning of the execution. The default size is 35MB but can easily be adjusted if needed. The allocated memory is in form of an array and is continuously being filled with random values at a user defined rate. [36, 6]

memoryleak.c

memoryleak.c iteratively generates new arrays with a user defined size at a random place in the computers memory and fills it step-by-step. The rate at which the arrays are filled can also be defined by the user. The allocated memory is not freed during the execution of the application. [36, 6]

networkcontention.c

This anomaly simulates contention between two nodes that are being connected by a network switch. Each node send messages to the other node. Additionally, the bandwitdth (the size of the messages) can be configured by the user. [36, 6]

4.5 Software Set-up

It was planned initially to install the 3 components prometheus-slurm-exporter, Prometheus and Grafana on an exclusive new node of the miniHPC, the monitoring node. However, there has been a disagreement as to where exactly the components should be installed. From a system administrators point of view, it does not make sense to install the components on the monitoring node as it can occur that nodes are subject to re-installation-processes that would be too time-consuming. Another problem is the fact that the monitoring node needs to be exposed to the internet in order to access the web-interface. As a temporary trade-off solution we have agreed to make use of a dedicated virtual machine, on which the web-based applications Prometheus and Grafana will be hosted and the Prometheus data will be stored. The slurm-exporter-script was installed directly onto the miniHPC where a dedicated port has been opened. The application has been deployed as a .service file, so that the prometheus-slurm-exporter script is persistently running as a daemon. In Figure 4.7 the reader can see the component diagram which provides an holistic overview of how the software is set-up.

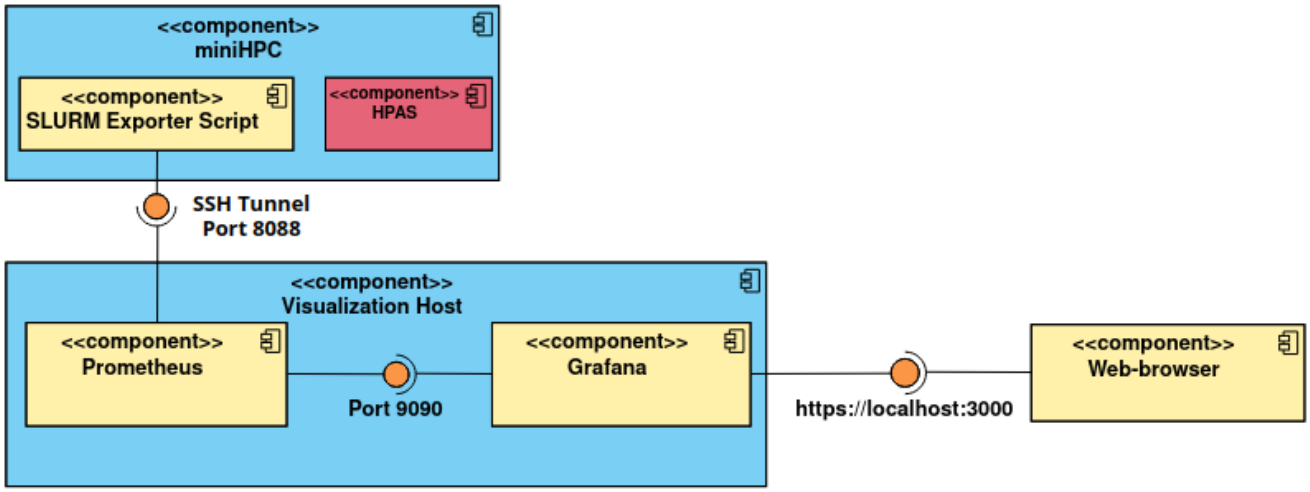

Figure 4.7: Component diagram of installed software

In order to establish a concise overview of how data can be displayed in Grafana the following time-sequence diagram can help the reader understanding workflow logic (Figure 4.8).

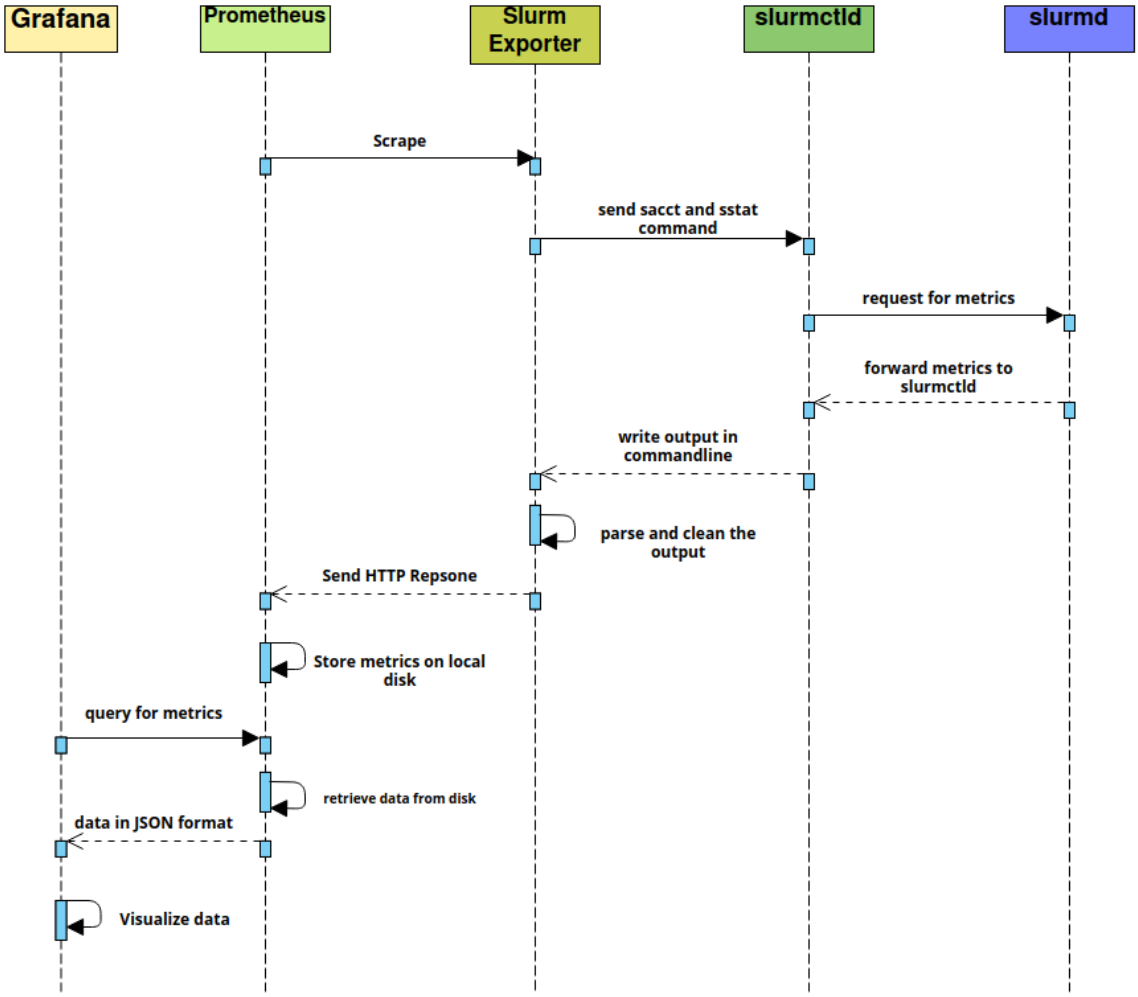

Figure 4.8: Sequence Diagram of the workflow

Chapter 5

Results

As a result of our work, software has been developed that defines new Prometheus metrics 8.4. Apart from that the installation process has been documented in 8. A slurm-exporter-daemon has been deployed on the miniHPC that continuously exports metrics to Prometheus.

5.1 What anomalies can be captured?

We carried out a set of tests with our developed monitoring set-up using the discussed anomaly suite (Chapter 4.4.1) to find out which anomalies can be captured. As a first step, it made sense to simply graph certain metrics as time-series that we expected to be related to the anomalies and analyse the properties of the plots. Afterwards, the task was to find the appropriate toolkit in Prometheus and Grafana in order to flag the specific anomalous behaviour of the corresponding jobs that would let us reach one of our research goals. Worth noting is that the techniques used for the detection of anomalous behaviour is adapted specifically to the anomaly suite and are therefore not generally valid. However, the results show that Prometheus can indeed flag processes that contain obvious synthetically ingested anomalies.

5.1.1 CPU anomalies

The HPAS anomaly cpuoccupy.c (Chapter 4.4.1) targets the CPU usage of the program. We came to the conclusion that anomalous program influences the average CPU Frequency as well as the CPU time of the program. The two prometheus queries used in order to detect these anomalies were therefore **"slurm_avecpufreq_runningjob"** and **"slurm_avecpu˙_runningjob"** (CPU Time) respectively. We have made the observation that the lower the cpu usage of the simulated program, the higher the standard-deviation of the cpu-frequency. This circumstance has been captured visually in Figure 5.1. Apart from influencing the cpu frequency, the anomaly also manifests itself in the CPU Time. We have noticed that the derivative of CPU Time corresponds accurately to the amount of CPU utilization specified (Figures, 5.1, 5.3).

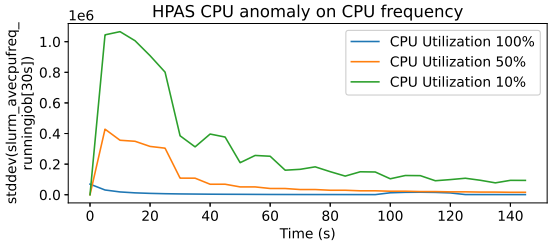

Figure 5.1: CPU anomaly on CPU frequency
y axis: Standard deviation
of CPU frequency of last 30 seconds monitored.

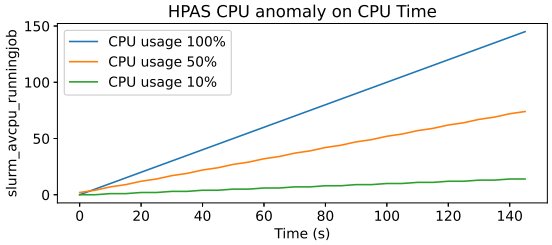

Figure 5.2: CPU anomaly on CPU Time
y axis: Accumulated CPU Time of program execution.

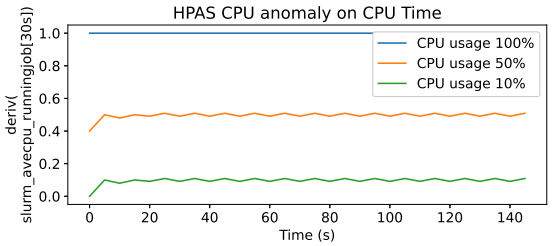

Figure 5.3: Derivative of CPU Time corresponds to CPU-Usage

5.1.2 I/O Bandwidth

As discussed in the methods section above (Chapter 4.4.1), HPAS provides three different programs for simulating I/O anomalies.

The properties of iobandwidth.sh is to increase the I/O bandwidth significantly. Therefore we expected the maximum amount of bytes read to increase. SLURM provides the user with the metric **maxdiskwrite** and **maxdiskread** which essentially captures the maximum number of bytes that have been written or read by a job. These metrics can be queried using **"slurm_maxdiskwrite_runningjob"** and **"slurm_maxdiskread_runningjob"** respectively. Essentially, the graph of the two metrics (Figure 5.4 and 5.5) can be visualized as monotonically increasing functions. A suited Prometheus tool to capture such an anomaly is to calculate the derivative of the respective time-series and specify an alert to be triggered should the it exceed a defined threshold.

In contrast to the iobandwidth.sh anomaly, iometadata.c does only create files and deletes them again. Consequently, we have noticed that the **maxdiskwrite** (Figure 5.6) increases for this anomaly whereas the bytes read were not affected 5.7).

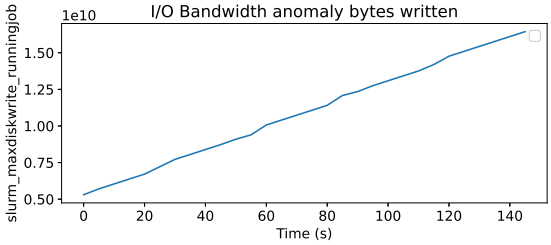

Figure 5.4: I/O bandwidth anomaly bytes written

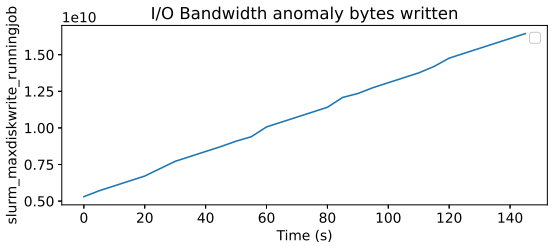

Figure 5.5: I/O bandwidth anomaly bytes read

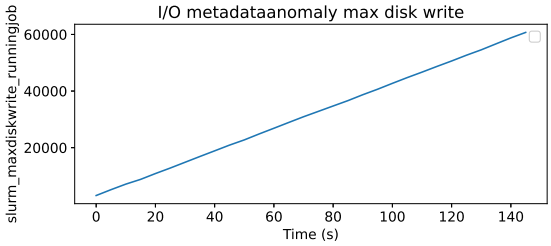

Figure 5.6: I/O metadataanomaly bytes written

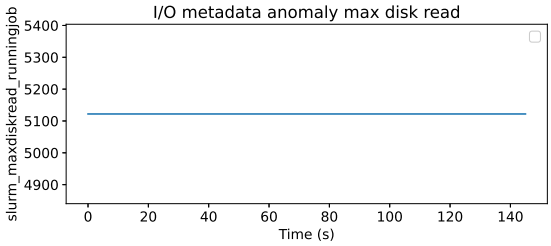

Figure 5.7: I/O metadataanomaly bytes read

5.1.3 Memory anomalies

We expected the memoryleak anomaly to manifest itself in increasing **maxvmsize** and **maxrss** which can be captured using the two slurm queries **"slurm_maxvmsize_runningjob"** and **"slurm_maxrss_runningjob"** respectively. Similarly to the I/O-bandwidth anomalies (Chapter 5.1.2), memoryleaks manifest themselves in monotonically increasing function plots (Figures 5.8, 5.9). We have therefore captured the anomaly with the same approach as the I/O-bandwidth anomaly, by defining alerts to be triggered should the derivative of the function exceed a gradient of 10 during the last 30 seconds of monitoring. The memoryeater.c anomaly could also be captured using these two metrics. We allocated 35mb to the running application which could easily be detected by a constant exceedingly high amount of **maxvmsize** and **maxrss** (Figures 5.10, 5.11).

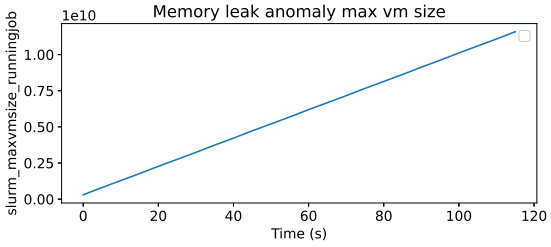

Figure 5.8: Memory leak anomaly max VM Size

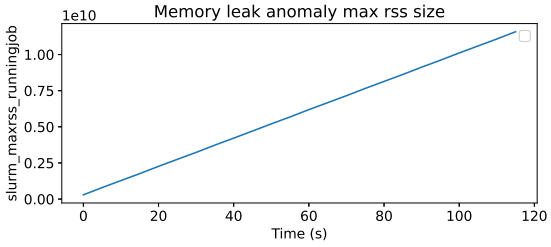

Figure 5.9: Memory leak anomaly max RSS

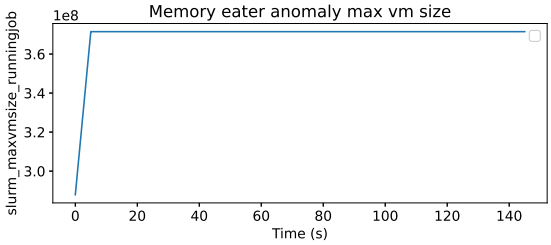

Figure 5.10: Memory eater anomaly max VM size

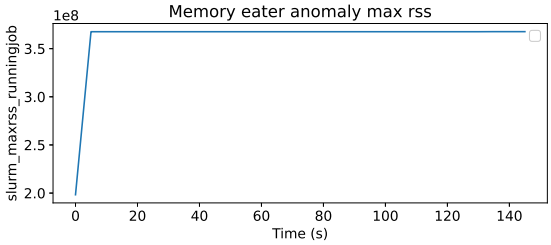

Figure 5.11: Memory eater anomaly max RSS

5.1.4 Load imbalance anomalies

SLURM does provide useful metrics in order to detect load-imbalance. The three metrics **tresusageinav, tresusageinmax and tresusageinmin**. As outlined in table 4.1, SLURM divides jobs into tasks which are distributed across the cluster. Each of these tasks has been granted a certain amount of CPU-Time by each timestep. **Tresusageinav, tresusageinmax and tresusageinmin** return the values for the average CPU-Time used for all task, as well as the maximum and minimal CPU time of a task. Ideally, the three values should be the same for each time-step as this implies that every task has been given exactly the same CPU-time.
These three metrics can be queried with **"slurm_tresusageinav_runningjob"**, **"slurm_tresusageinmax_runningjob"** and **"slurm_tresusageinmin_runningjob"** respectively. By taking the difference between the average and minimal CPU-Time of the tasks, a time-series is produced that gives an indication if tasks are being executed with the same computational effort. We have used the load-imbalanced mandel-brot program in the reference plot. As can be seen on Figure 5.12, the metric **(slurm_tresusageinmax_runningjob -slurm_tresusageinav_runningjob)** produces a strongly oscillating plot, which implies that for a lot of time-steps, there is a difference between the average and maximum CPU-Time between two tasks. As a reference, a load balanced application (hpl-8) produces a mostly constant plot of the same metric. In figure 5.13 we have summed up the time-differences between the average and minimal case using **sum_over_time((slurm_tresusageinmax_runningjob- slurm_tresusageinav_runningjob)[1d:5s])**
Another possible indication of load imbalance can be determined if we look at the node that has been granted the most TRES resources. In the load-imbalanced application mandel.c one node is continously being granted the most resources (mandel.c has been run on two nodes), which signals load-imbalance. This was queried with **slurm_tresusageinmaxnode_runningjob**. Figure 5.14 contains the reference plot.

5.1.5 Limitations of anomaly detection

Some anomalies generated by HPAS could not be detected. For one, the memorybw.c anomaly was not capturable as SLURM does not provide the means to monitor stack memory during the execution of a program. Furthermore network contention could not be detected as the SLURM client installed on miniHPC does not contain the necessary plugins to monitor network interfaces (such as [29]). Unfortunately cache copy events could also not be captured as SLURM does not provide the tools necessary.

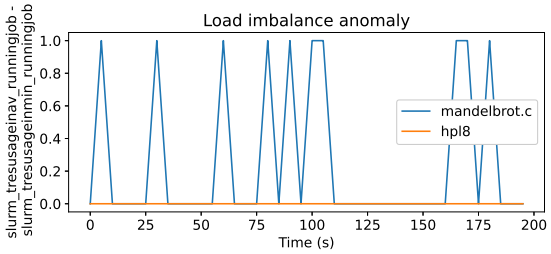

Figure 5.12: Load imbalance anomaly
y axis: slurm_tresusageinav_runningjob
-slurm_tresusageinmin_runningjob

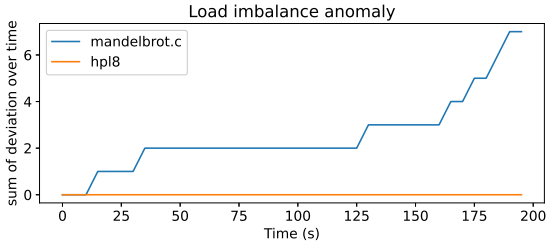

Figure 5.13: Load imbalance anomaly
y axis: sum_over_time((slurm_tresusageinmax_runningjob-
slurm_tresusageinav_runningjob)[1d:5s])

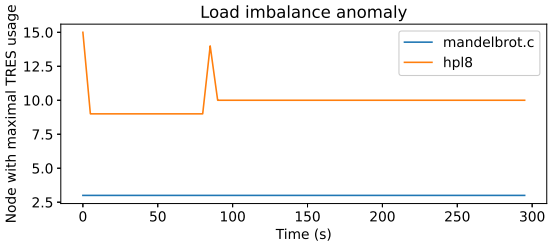

Figure 5.14: Load imbalance anomaly, Node with
maximum TRES usage

5.1.6 Summary

This table contains a concise overview as to which experiments were executed with regards to the experiments and results.

Anomaly Table				
Anomalous program	**Anomalous characteristics**	Anomaly capturable?	**Prometheus Query used**	**Alert-condition**
cachecopy.c	Cache read and write	✗	-	-
cpuoccupy.c	High / Low CPU Usage	✓	slurm_avecpufreq_runningjob slurm_avecpu_runningjob	standard deviation >= 1000 for last 30s derivative <= .95
iobandwidth.sh	Read and write files	✓	slurm_maxdiskwrite_runnnigjob slurm_maxdiskread_runningjob	derivative >= 10 for last 30s
iometadata.c	create and delete files	✓	slurm_maxdiskwrite_runnnigjob	derivative >= 10 for last 30s
membw.c	Stack memory write	✗	-	-
memeater.c	Allocates, fills and releases memory	✓	slurm_maxvmsize_runningjob slurm_maxrss_runningjob	maxrss or maxvm constantly above 20mb
memleak.c	Allocate and fill memory	✓	slurm_maxvmsize_runningjob slurm_maxrss_runningjob	derivative >= 10 for last 30s
netoccupy.c	sends messages between nodes	✗	-	-
mandel.c	Load imbalance	✓	slurm_tresusageinmaxnode_runningjob slurm_tresusageinav_runningjob slurm_tresusageinmin_runningjob slurm_tresusageinmax_runningjob	Sum of difference between slurm_tresusageinav_runningjob and slurm_tresusageinmin_runningjob exceeds 5 seconds

Table 5.1: Anomaly Table

5.2 Resulting Grafana visualization

In the following the reader can find the Grafana visualizations created using the methodolody discussed in chapter 4.

jobs	jobname (last)	username (last)	CPU Frequ	CPU Anc	MaxVMSize	Memo	Maxdiskwrite	Maxdiskread	I/O Anor
790080	hpl-8		996	Normal	3754790912	Normal	259000	11820140	Norma
790084	interactive		88189	Anomaly	4174491648	Normal	40014	4758200	Norma
790000	mandel_no_sync		5152	Normal	3118014464	Normal	1881	11762542	Norma
790001	mandel_no_sync		6169	Normal	3118149632	Normal	1881	11762542	Norma
790002	mandel_no_sync		2481	Normal	3118010368	Normal	1881	11762548	Norma
790003	mandel_no_sync		13978	Normal	3118014464	Normal	1881	11762548	Norma
790004	mandel_no_sync		12606	Normal	3118018560	Normal	1881	11762548	Norma
790005	mandel_no_sync		2114	Normal	3118014464	Normal	1881	11762548	Norma
790006	mandel_no_sync		12020	Normal	3118018560	Normal	1881	11762548	Norma
790007	mandel_no_sync		4611	Normal	3118018560	Normal	1881	11762548	Norma
790008	mandel_no_sync		2228	Normal	3118018560	Normal	1881	11762548	Norma
790009	mandel_no_sync		15402	Normal	3118145536	Normal	1881	11762548	Norma

Figure 5.15: Job overview table

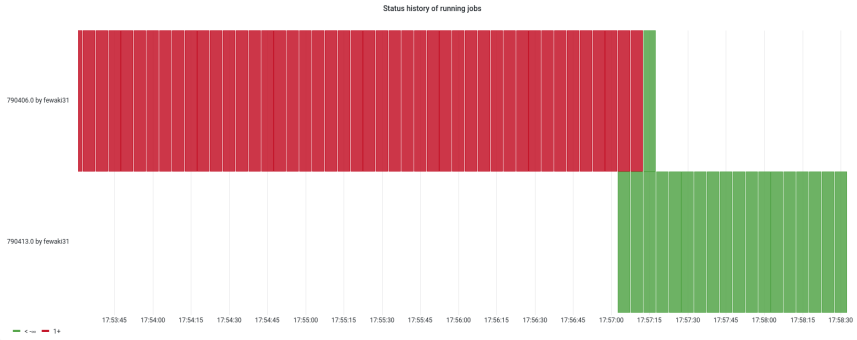

Figure 5.16: Job status history

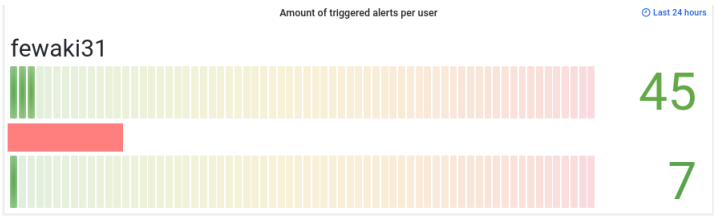

Figure 5.17: Amount of triggered alerts in per user

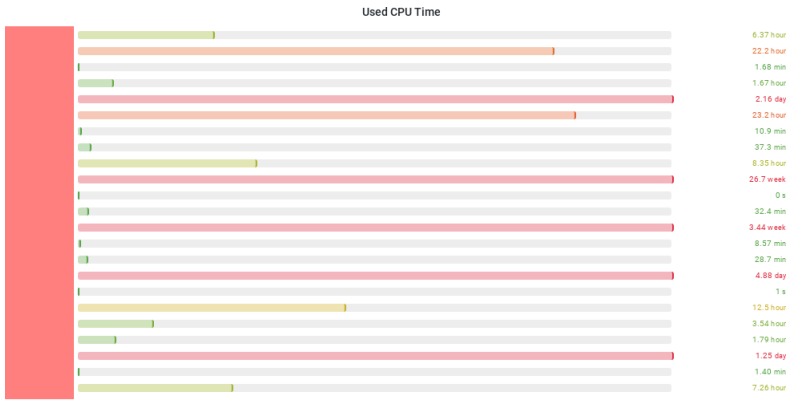

Figure 5.18: Used CPU-Time per user (anonymized)

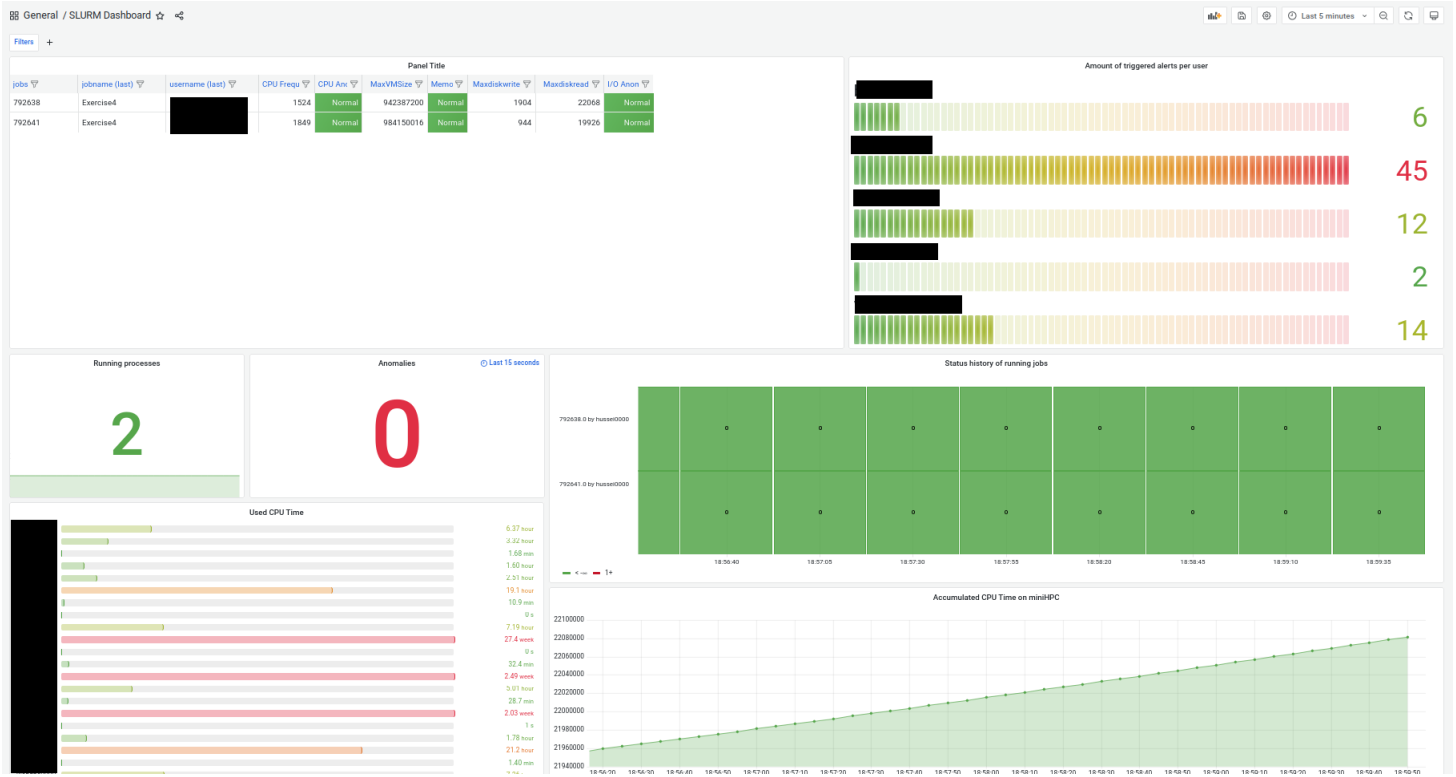

Figure 5.19: Dashboard created anonymized

Chapter 6

Discussion

6.1 Interpretation of the results

The result of this thesis is the newly configured monitoring framework.
Previously, with the ganglia monitoring framework, job-level monitoring was not possible. Additionally, users were unable to specify queries and to configure the properties of the plots shown. Users are now able to query for over 20 metrics of running jobs provided by the workload manager SLURM. With the query-power of the querylanguage PromQL and the visualization power of Grafana, this opens up the door for various interesting visualizations which increases the observability of the miniHPC with the potential to increase the efficiency.

Furthermore, since data is now stored in a compressed and memory efficient database, we have achieved the storage of valuable data that can be subject to further research for instance for training artificial intelligence to detect anomalous behaviour in jobs.

This work also can be used as a proof of concept that the monitoring set-up can be used in order to capture some anomalies. Additionally, there now exists a concise overview of jobs that are currently running. Jobs with obvious anomalies provided from an anomaly suite are flagged automatically in an alert-table and a job-overview table. This gives system administrators and researchers the appropriate toolkit in order to detect and possibly cancel jobs with certain anomalies.
As the developed code for the data-cleaning and forwarding to HTTP is adaptable to other metric interfaces, it can easily be extended to capture further interesting metrics which is valuable in the sense that it can make the full monitoring infrastructure more powerful.

6.2 How do we compare to other related work?

Our solution approach was based on the PIKA [39] monitoring framework. As discussed in the related work section, there exists monitoring frameworks that do consider the topology of the cluster and can visualize how the processes are distributed across the HPC system [47]. It was initially planned to include a similar functionality in our monitoring tool using the Grafana utility 'node graphs' where nodes are an actual abstraction of computing nodes and the edges are the

interconnects between them. However, the node graphs are only supported for the data source XRay and not the Prometheus database [10]. Other approaches additionally are able to predict wait times of jobs that are in the pipeline such as [47].

6.3 Are there problems or limitations remaining?

In order to execute the sstat and sacct commands, slurmd sends requests to slurmctld daemon in order to query for metrics [45]. SLURM explicitly warns for performance variations in the slurmctld daemon that might occur when sending queries too frequently [31, 30]. This has the unfortunate implication that our monitoring framework is limited in the resolution of collected data of SLURM. Scraping with a too high frequency might even cause the slurmctld daemon to deny service which implies that no more jobs can be scheduled and no more resources can be monitored. In order to prevent potential system downtime we have decided to set the scraping frequency of Prometheus to 5 seconds which did not cause any problems for the duration of the monitoring. It might even be possible to collect data with a lower resolution, however, but we deemed the risk of causing system downtime as too big.
Another limiting factor related to this problem comes from the open-source exporter prometheus-slurm-exporter [23]. In fact, the code provided from the github repository [23] does not take into account efficiency and therefore sends too many SLURM queries that could be implemented more efficiently. Consequently we have decided to only use our code, that targets metrics coming from running jobs and that is more efficient.

Additionally, it is worth noting that the detection of anomalies is highly adapted to the properties of the anomalous jobs provided from the HPAS anomaly suite and is therefore not generally valid.

Due to the constraint of time and other resources, Prometheus and Grafana could not be installed directly on the miniHPC's monitoring node. However as by the termination of this thesis, no solution has been determined yet. This issue is subject to further discussion.

Chapter 7

Conclusion

7.1 What did we achieve?

In this thesis a new monitoring framework for the HPC Groups miniHPC has been configured that consists of the software SLURM, Prometheus and Grafana. We have used SLURM in order to collect metrics about running jobs and developed Go Code in order to parse, clean and forward data to Prometheus, the monitoring tool that contains a time-series database. A SLURM exporter service has been installed on a dedicated port of the miniHPC that collects job data of every user with a resolution of 5 seconds. Furthermore, we have defined a set of new Prometheus database queries with which data can be retrieved. We have put our new monitoring set-up through its paces by evaluating what Prometheus and Grafana utilities can be used in order to automatically detect anomalous behaviour of running jobs. Using the HPAS [36] anomaly suite we have simulated various anomalous jobs of which we were able to flag memory-leaks, CPU-anomalies, I/O bandwidth anomalies. Moreover, we have used Grafana's tools in order to create concise overviews such as tables, time-series graphs alert-lists and bar gauges that allow for the rapid detection of anomalous behaviour. We can therefore conclude that SLURM in combination with Grafana and Prometheus is suitable for the detection of anomalies.

7.2 Future work

In order to make the detection of anomalies more generally valid, more research needs to be put into the study of anomalous behaviour of jobs. For the detection of anomalies in time-series data there exists approaches for instance by generating predictive models of the development of metrics over the execution of a program by using historical data. Outlier data points that have an obvious deviation from the predictive model can be flagged as anomalies at run-time of the program. Building such predictive models requires collecting more data of running jobs, preferably with a high resolution that should be stored in a time-series database such as Prometheus. For our monitoring infrastructure an approach for getting such predictive models is to use Google's Big Query ML [35] Grafana plug-in [3] which relies on historical Prometheus time-series data. Even-though we have collected a variety of metrics that help in the detection of anomalies, SLURM does provide more useful metrics, some of which we were unable to capture as the corresponding plug-ins are not in-

stalled. In the future, it might be desirable to also collect metrics about the network bandwidth of the cluster, energy consumption and also capture cache-level metrics. This data could be collected in the future by installing the SLURM HDF5 [29] plugin. Grafana provides more visual tool-kits that might be interesting for the analysis of the network topology of the miniHPC. For instance, node graphs can be be used in order to display on what nodes jobs are currently distributed.

Bibliography

[1] About grafana. https://grafana.com/oss/grafana/. Accessed: 2022-03-07.

[2] Angular webframework. https://angular.io/, note = Accessed: 2022-02-17.

[3] Big query ml plug-in for grafana. https://grafana.com/grafana/plugins/grafana-bigquery-datasource/, note = Accessed: 2022-02-17.

[4] Client libraries for prometheus. https://prometheus.io/docs/instrumenting/clientlibs/, note = Accessed: 2022-02-17.

[5] Energy efficient hpc working group. https://eehpcwg.llnl.gov/. Accessed: 2022-02-17.

[6] Github of hpas performance anomaly suite. https://github.com/peaclab/HPAS. Accessed: 2022-03-07.

[7] The go language. https://go.dev/. Accessed: 2022-09-05.

[8] Grafana. https://grafana.com/. Accessed: 2022-02-17.

[9] Grafana bar gauge. https://grafana.com/docs/grafana/latest/visualizations/bar-gauge-panel/.

[10] Grafana node graph data-source. https://community.grafana.com/t/datasource-for-node-graph/43944, note = Accessed: 2022-04-17.

[11] Grafana status history. https://grafana.com/docs/grafana/latest/visualizations/status-history/.

[12] Grafana table. https://grafana.com/docs/grafana/latest/visualizations/table/, note = Accessed: 2022-02-17.

[13] Influx database. https://www.influxdata.com/, note = Accessed: 2022-02-17.

[14] Install on rpm-based linux (centos, fedora, opensuse, red hat). https://grafana.com/docs/grafana/latest/installation/rpm/. Accessed: 2022-02-17.

[15] Install prometheus server on centos 7 / ubuntu 18.04. https://computingforgeeks.com/install-prometheus-server-on-ubuntu-centos/. Accessed: 2022-02-17.

[16] Mini hpc. https://hpc.dmi.unibas.ch/en/research/minihpc/. Accessed: 2022-02-17.

[17] Mini hpc old monitoring ui. https://hpc.dmi.unibas.ch/en/research/minihpc/. Accessed: 2022-02-17.

[18] Monitoring and operational data analytics conference. https://moda20.sciencesconf.org/resource/page/id/3. Accessed: 2022-03-07.

[19] Openmp. https://www.openmp.org/. Accessed: 2022-03-15.

[20] Prometheus exporter. https://prometheus.io/docs/instrumenting/exporters/#http. Accessed: 2022-09-05.

[21] Prometheus query functions. https://prometheus.io/docs/prometheus/latest/querying/functions/#aggregation_over_time, note = Accessed: 2022-02-17.

[22] Prometheus scraping mechanism. https://prometheus.io/docs/prometheus/latest/getting_started/#starting-prometheus. Accessed: 2022-09-05.

[23] Prometheus slurm exporter github. https://github.com/vpenso/prometheus-slurm-exporter, note = Accessed: 2022-02-17.

[24] Prometheus soundcloud blog. https://developers.soundcloud.com/blog/prometheus-monitoring-at-soundcloud. Accessed: 2022-02-17.

[25] Prometheus storage. https://prometheus.io/docs/prometheus/latest/storage/#local-storage. Accessed: 2022-09-05.

[26] The prometheus time series database talk. https://promcon.io/2016-berlin/talks/the-prometheus-time-series-database/, note = Accessed: 2022-02-17.

[27] Round robin database tool. https://oss.oetiker.ch/rrdtool/tut/rrdtutorial.en.html. Accessed: 2022-03-14.

[28] slurm design. https://slurm.schedmd.com/slurm_design.pdf. Accessed: 2022-02-17.

[29] Slurm hdf5 plugin. https://slurm.schedmd.com/hdf5_profile_user_guide.html.

[30] Slurm sacct command. https://slurm.schedmd.com/sacct.html, note = Accessed: 2022-02-17.

[31] Slurm sstat command. https://slurm.schedmd.com/sstat.html, note = Accessed: 2022-02-17.

[32] Slurmvision. https://prometheus.io/docs/introduction/overview/. Accessed: 2022-02-17.

[33] sstat command. https://slurm.schedmd.com/sstat.html. Accessed: 2022-03-14.

[34] Top 500 supercomputer. https://www.top500.org/statistics/. Accessed: 2022-03-07.

[35] What is big query ml. https://cloud.google.com/bigquery-ml/docs/introduction, note = Accessed: 2022-02-17.

[36] Emre Ates, Yijia Zhang, Burak Aksar, Jim Brandt, Vitus Leung, Manuel Egele, and Kaan Cokun. Hpas: An hpc performance anomaly suite for reproducing performance variations. *ICPP 2019: Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 08 2019.

[37] Norman Bourassa, Walker Johnson, Jeff Broughton, Deirdre Carter, Sadie Joy, Raphael Vitti, and Peter Seto. Operational data analytics: Optimizing the national energy research scientific computing center cooling systems. pages 1–7, 08 2019.

[38] B. Brazil. *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring*. O'Reilly Media, 2018.

[39] Robert Dietrich, Frank Winkler, Andreas Knpfer, and Wolfgang Nagel. Pika: Center-wide and job-aware cluster monitoring. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 424–432, 2020.

[40] Jan Eitzinger, Thomas Gruber, Ayesha Afzal, Thomas Zeiser, and Gerhard Wellein. Clustercockpit a web application for job-specific performance monitoring. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–7, 2019.

[41] Robert L. Henderson. Job scheduling under the portable batch system. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 279–294, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[42] Bruce Hendrickson and Jonathan W. Berry. Graph analysis with high-performance computing. *Computing in Science Engineering*, 10(2):14–19, 2008.

[43] G. S. Hodgson, P. Dzwig, H. M. Liddell, and D. Parkinson. Application of hpc to medium-size stochastic systems with non-linear constraints in finance. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, pages 411–418, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[44] Thomas Jakobsche, Nicolas Lachiche, Aurlien Cavelan, and Florina M. Ciorba. An execution fingerprint dictionary for hpc application recognition, 2021.

[45] Morris A. Jette, Andy B. Yoo, and Mark Grondona. Slurm: Simple linux utility for resource management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, pages 44–60. Springer-Verlag, 2002.

[46] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[47] Ashish Pal and Preeti Malakar. Map: A visual analytics system for job monitoring and analysis. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 442–448, 2020.

[48] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proc. VLDB Endow.*, 8(12):18161827, aug 2015.

[49] Thomas Rhl, Jan Eitzinger, Georg Hager, and Gerhard Wellein. Likwid monitoring stack: A flexible framework enabling job specific performance monitoring for the masses. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 781–784, 2017.

[50] Erik Schnetter, Marek Blazewicz, Steven R. Brandt, David M. Koppelman, and Frank Löffler. Chemora: A PDE solving framework for modern HPC architectures. *CoRR*, abs/1410.1764, 2014.

[51] Thomas Sterling, Matthew Anderson, and Maciej Brodowicz. *High Performance Computing*. Morgan Kaufmann, Boston, 2018.

Chapter 8

Appendix

This document is aimed at providing an detailed process description for the installation of Grafana, Prometheus and the Slurm exporter Script. The installation has been tested on CentOS Linux 7 (Operating System of the MiniHPC).

8.1 Installation of Prometheus

The following installation documentation has been taken from [15].

- Add a Prometheus subgroup:

```
sudo groupadd --system prometheus
```

- Creating a Prometheus system user :

```
sudo useradd -s /sbin/nologin --system -g prometheus prometheus
```

- Creating a directory for Prometheus, where data will be stored.

```
sudo mkdir /var/lib/prometheus
```

- Creating configuration directories for Prometheus.

```
for i in rules rules.d files_sd; \
do sudo mkdir -p /etc/prometheus/${i}; done
```

- Installing wget.

```
sudo yum -y install wget
```

- Create a directory and navigate to it:

```
mkdir -p /tmp/prometheus && cd /tmp/prometheus
```

- Download the distribution from github:

```
curl -s https://api.github.com/repos/prometheus\
/prometheus/releases/latest \
  | grep browser_download_url \
  | grep linux-amd64 \
  | cut -d '"' -f 4 \
  | wget -qi -
```

- extract the zipped file:

```
tar xvf prometheus*.tar.gz
```

- change directory to the extracted folder:

```
cd prometheus*/
```

- Move the prometheus binary files to /usr/local/bin/

```
sudo mv prometheus promtool /usr/local/bin/
```

- Move prometheus configuration template to /etc directory.

```
sudo mv prometheus.yml  /etc/prometheus/prometheus.yml
```

- Create/Edit a Prometheus configuration file

```
sudo vim /etc/prometheus/prometheus.yml
```

- Create a Prometheus systemd Service unit file

```
sudo vim /etc/systemd/system/prometheus.service
[Unit]
Description=Prometheus
Documentation=https://prometheus.io/docs/introduction/overview/
Wants=network-online.target
After=network-online.target

[Service]
Type=simple
Environment="GOMAXPROCS=1"
User=prometheus
Group=prometheus
ExecReload=/bin/kill -HUP $MAINPID
ExecStart=/usr/local/bin/prometheus \
  --config.file=/etc/prometheus/prometheus.yml \
  --storage.tsdb.path=/var/lib/prometheus \
  --web.console.templates=/etc/prometheus/consoles \
  --web.console.libraries=/etc/prometheus/console_libraries \
```

```
  --web.listen-address=0.0.0.0:9090 \
  --web.external-url=

SyslogIdentifier=prometheus
Restart=always

[Install]
WantedBy=multi-user.target
```

- Important!: The line

  ```
  Environment ="GOMAXPROCS=1"
  ```

  needs to be edited with the number of virtual vcpus. This can be queried in a Linux environment with

  ```
  cat /proc/cpuinfo | grep processor | wc -l
  ```

  In case of the MiniHPC there is 20 vcpus.

- Change the ownership of this of prometheus folder to Prometheus user and group.

  ```
  sudo chown -R prometheus:prometheus /var/lib/prometheus/ -l
  ```

- Reload the systemd daemon and start the service and configure prometheus to start at boot.

  ```
  sudo systemctl daemon-reload
  sudo systemctl start prometheus
  sudo systemctl enable prometheus -l
  ```

- Check if Prometheus is in fact running:

  ```
  systemctl status prometheus -l
  ```

- Open a port on the firewall:

  ```
  sudo firewall-cmd --add-port=9090/tcp --permanent
  sudo firewall-cmd --reload -l
  ```

- Now Prometheus can be launched in a browser by accessing:

  ```
  http://localhost:9090
  ```

8.2 Installation of Grafana

The installation documentation is taken from [14]

- Create a repo file

  ```
  sudo nano /etc/yum.repos.d/grafana.repo
  ```

- Update the repo file with the following text:

  ```
  [grafana]
  name=grafana
  baseurl=https://packages.grafana.com/oss/rpm
  repo_gpgcheck=1
  enabled=1
  gpgcheck=1
  gpgkey=https://packages.grafana.com/gpg.key
  sslverify=1
  sslcacert=/etc/pki/tls/certs/ca-bundle.crt -l
  ```

- Install Grafana with yum

  ```
  sudo yum install grafana
  ```

- reload daemon and start the grafana server.

  ```
  sudo systemctl daemon-reload
  sudo systemctl start grafana-server
  ```

- check if grafana-server is running

  ```
  sudo systemctl status grafana-server
  ```

- Configure Grafana to start at boot.

  ```
  sudo systemctl enable grafana-server
  ```

- Access grafana on localhost

  http://localhost:3000/

8.3 Prometheus YAML file

```
global:
  scrape_interval: 15s
  evaluation_interval: 15s
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
    - static_configs:
        - targets:
          # - alertmanager:9093

# Load rules once and periodically evaluate them according
#to the global 'evaluation_interval'.
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:

#
# SLURM resource manager:
#
  - job_name: 'my_slurm_exporter'

    scrape_interval:  5s

    scrape_timeout:   5s

    static_configs:
      - targets: ['0.0.0.0:8088']


  # The job name is added as a label `job=<job_name>` to any timeseries
  #scraped from this config.
  - job_name: "prometheus"

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
```

```
- targets: ["localhost:9090"]
```

Listing 8.1: Prometheus YAML configuration file

8.4 Added Prometheus metrics

Prometheus Query	Metric
slurm_allocnodes_runningjob	Allocated nodes for job
slurm_maxrss_runningjob	Max RSS of job
slurm_avecpu_runningjob	Accumulated CPU Time of job
slurm_avediskread_runningjob	Bytes read by job on average
slurm_maxdiskread_runningjob	Bytes read by job at maximum
slurm_maxdiskwrite_runningjob	Bytes written by job at maximum
slurm_maxvmsize_runningjob	MaxVmsize of job
slurm_avecpufreq_runningjob	Average CPU frequency
slurm_avediskwrite_runningjob	Bytes written on average
slurm_avepages_runningjob	Average number of pages
slurm_averss_runningjob	Average resident set size
slurm_maxdiskreadnode_runningjob	Node that reads the most bytes
slurm_maxdiskreadtask_runningjob	Task that reads the most bytes
slurm_maxdiskwritenode_runningjob	Node that writes the most bytes
slurm_maxdiskwritetask_runningjob	Task that writes the most bytes
slurm_maxpagesnode_runningjob	Maximum number of pages per node
slurm_maxpagestask_runningjob	Maximum number of pages per node
slurm_mincpu_runningjob	Minimal CPU Time of a task
slurm_mincpunode_runningjob	Node with the least CPU usage
slurm_mincputask_runningjob	Task with the least CPU usage
slurm_ntask_runningjob	Number of tasks in job
slurm_tresusageinav_runningjob	Average TRES usage per task
slurm_tresusageinmax_runningjob	Maximum TRES usage per task
slurm_tresusageinmin_runningjob	Minimum TRES usage per task
slurm_tresusageinmaxnode_runningjob	Node with most TRES usage

University of Basel

Faculty of Science

Declaration on Scientific Integrity
(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis: HPC Job-Monitoring with SLURM, Prometheus and Grafana

Name Assessor: Prof. Dr. Florina Ciorba

Name Student: Pascal Kunz

Matriculation No.: 2018-058-529

With my signature I declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Place, Date: Birsfelden, 14th May, 2022 Student: _____

Will this work, or parts of it, be published?

- ○ No
- ◉ Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: 16th May, 2022

Place, Date: Birsfelden, 14th May, 2022 Student: _____

Place, Date: _____ Assessor: _____

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis