Seminar Report

Scalability Evaluation of Prometheus for HPC Monitoring

Lars Quentin

MatrNr: 21774184

Supervisors: Marcus Merz, Prof. Dr. Julian Kunkel

Georg-August-Universität Göttingen
Institute of Computer Science

February 14, 2024

Abstract

In order to manage multiple systems, especially large High-Performance Computing (HPC) clusters, proper utilization of monitoring is of utmost importance. Monitoring has several use cases, from providing basic health overviews to understanding usage patterns or doing demand analysis by forecasting utilization based on current data. At the GWDG, two different Grafana based monitoring systems are used, one based on the InfluxDB Time series database (TSDB) and one relying on Prometheus as the data source.

This report provides a methodology for benchmarking pull-based monitoring systems such as Prometheus on various metrics. Additionally, it provides an answer on whether both monitoring systems can be merged by evaluating the viability and scalability of a Prometheus based monitoring system for a realistic HPC use case.

To accomplish this goal, four different benchmarks were designed: The first benchmark measures the performance of the metric gathering routine by patching the collector. The next benchmark provides a end to end measurement of the scalability and resilience of the collector utilizing traditional HTTP load generator technologies. The last collector benchmarks captures the performance penalty incurred by the metric collector on classical HPC computation load by analyzing the jitter between time measurements. Lastly, the final benchmark analyzes the overall performance and scalability of Prometheus itself by mocking the collector daemons and counting the number of pulls.

The results show that, although the collector did not handle the stress test load well, its throughput and overall performance is sufficient and the collection time acceptable, even using only a single process for execution. Prometheus itself did not scale well beyond a certain amount of nodes, failing with 2000 collectors when all run on the same hardware. Note that this is not merely a result of underprovisioned hardware but instead most likely an architectural problem, since all measurable metrics show that the server was not fully utilized.

While this report presents a design and implementation of a performance penalty benchmark, the results were, mostly due to the sheer size of the data set, only partly analyzable within the scope of this report. The data shows a visibly longer tail in execution time, affirming that a measuable performance penalty exists. To provide a definite, quantifiable conclusion on the performance penalty created by the collector, further analysis has to be done.

Concluding this report, while the collectors performance is sufficient, Prometheus performance strongly decreases once a certain amount of nodes are reached, implying that it is not usable for very large HPC clusters without the usage of Prometheus clustering techniques.

Statement on the usage of ChatGPT and similar tools in the context of examinations

In this work I have used ChatGPT or a similar AI-system as follows:

- ☐ Not at all
- ☐ In brainstorming
- ☐ In the creation of the outline
- ☐ To create individual passages, altogether to the extent of 0% of the whole text
- ☐ For proofreading
- ☑ Other, namely: GitHub Copilot as coding assistant

I assure that I have stated all uses in full.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables iv

List of Figures iv

List of Listings iv

List of Abbreviations v

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Goals and Contributions . . . 2
  - 1.2.1 Structure . . . 2

2 Background: Monitoring 2
- 2.1 Current Architecture: GWDG SCC . . . 3
- 2.2 Current Architrecture: HLRN Emmy . . . 4
- 2.3 Prometheus . . . 5
- 2.4 TSDB Performance Evaluation . . . 5

3 Benchmark Methodology 6
- 3.1 Setup . . . 6
- 3.2 `node_exporter` . . . 7
  - 3.2.1 Metric Gathering . . . 7
  - 3.2.2 End to End . . . 8
  - 3.2.3 Jitter . . . 9
- 3.3 Prometheus . . . 10

4 Results 11
- 4.1 `node_exporter` Performance . . . 12
- 4.2 `node_exporter` Performance Penalty (Jitter) . . . 14
- 4.3 Prometheus . . . 15

5 Discussion 16

6 Conclusion 17
- 6.1 Future Work . . . 17

References 19

A Example Autogenerated Prometheus Config A1

B Prometheus Node Utilization data A1
- B.1 I/O was not the bottleneck . . . A2
- B.2 Memory was not the bottleneck . . . A3
- B.3 CPU was not the bottleneck . . . A5

List of Tables

List of Figures

1 The InfluxDB-based push-setup for the SCC. 4
2 The Prometheus-based pull-setup for Emmy. 4
3 Prometheus high level architecture [5]. 5
4 A kernel density estimation of the results. a) showing the raw times it took to collect all metics 1000 times. b) calculated the approximated requests per second from the initial data. 12
5 The requests per second given the amount of processes available for Gos runtime. Note that $n = 1$ failed while benchmarking due to an error in the benchmark pipeline. 13
6 a) shows the average throughput and b) the percentage of requests which timed out. 13
7 The results for single nodes, plotted onto a 25 bin, linearly sized histogram. 14
8 Scaling up the number of mock nodes. The number of requests are the number of times Prometheus contacted one of the mock clients. All mock clients ran on a single node. 15
9 Scaling up the number of mock nodes. The number of requests are the number of times Prometheus contacted one of the mock clients. The mock clients were split onto two nodes. 16
10 The I/O wait times recorded, showing that there was no increasing queue of I/O operations. A2
11 The free memory, showing that there was more than enough free memory, and that the memory usage was not significantly different between the benchmarks. A3
12 The total non-used memory (including page cache and buffer cache), again showing no significant difference between the well performing 1500 node benchmark and the slow 2000 node benchmark A4
13 No swap was used at both benchmarks. A4
14 The accumulated CPU usage, both system time (sy) and user time (us), showing that the processor was not fully utilized in both benchmarks, and that due to the high amount of threads available the CPU usage was not significantly different. A5

List of Listings

1 How the metrics are collected in `collector/collector.go` 7
2 The patched collector measuring the collection time. Note that the `RealCollect` function contains the same code as the unpatched `Collect`. 8
3 A simplified C-style pseudo code of the jitter program. 10
4 Example Autogenerated Prometheus Config (target list truncated and reformatted) A1

List of Abbreviations

ASGI Asynchronous Server Gateway Interface

CNCF Cloud Native Computing Foundation

GWDG Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

HPC High-Performance Computing

SCC Scientific Compute Cluster

TSBS Time Series Benchmark Suite

TSDB Time series database

1 Introduction

1.1 Motivation

With the rise of simulation and big data analytics as new foundations of all modern science, access to large scale HPC systems has become paramount for good research. Furthermore, as the area of machine and deep learning progresses, with models containing over one billion parameters [1], local computation becomes everless feasable, resulting in the rise of HPC usage amongst various domains.

This report is written as an addition to my student research at the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG). The GWDG is the data center of the University of Göttingen and the data and IT competence center of the Max Planck Society with its own HPC department, running four large scale HPC systems: The Scientific Compute Cluster (SCC), Emmy, Grete, and CARO:

- **SCC**[2]: The SCC is the oldest cluster maintained by the GWDG. Its user group is comprised of all researchers of the Max Planck Society as well as all (student) researchers of the University of Göttingen. It is a very heterogenous system, based on several different CPUs and GPUs generations, located over several locations within Göttingen. In total, it has 18,376 CPU Cores, 99 TB RAM and 5.2 PiB of Storage.
- **Emmy**[3]: Together with the Lise[1] cluster provided by the ZIB, Emmy is part of the fourth supercomputer generation of the HLRN[2]. It can be used by all HLRN and NHR users. Ranking 133th at the TOP500[3], Emmy is the biggest cluster hosted by the GWDG. It consists of 111,464 CPU Cores distributed over 1423 compute nodes resulting in a total peak compute power of 5.95 PetaFLOP/s.
- **Grete**: Grete is a GPU HPC cluster, formally part of the NHR system. It features 158 nodes with 2 AMD Epyc 7513 CPUs and four NVIDIA A100 GPUs each, connected through fast Infiniband fabric. As of November 2023, Grete is rank 142 of the TOP500.
- **CARO**[4]: Analagously to Emmy provided for the NHR, CARO is a compute cluster hosted for the DLR[4]. With its 1364 compute nodes with 175,744 CPU Cores and 3.46 PetaFLOP/s it ranks 228th on TOP500.

As the sheer number of nodes makes individually inspecting each one impossible, a centralized monitoring solution is required. Beyond getting an basic understanding on which node is alive, monitoring systems serve several important purposes. With an aggregated view, system admins can understand the usage patterns. Furthermore, it can be used as a means of load balancing for detecting preventable bottlenecks such as suboptimal job queue usage. Additionally, monitoring allows for better demand analysis and forecasting, allowing for more efficient, just in time hardware upgrades.

[1] `https://www.zib.de/research_services/supercomputing`
[2] Norddeutscher Verbund für Hoch- und Höchstleistungsrechnen `https://www.hlrn.de/`
[3] As of November 2023.
[4] Deutsches Zentrum für Luft- und Raumfahrt `https://www.dlr.de/en`

1.2 Goals and Contributions

As the time of this writing, the GWDG has two different, Grafana-based monitoring solutions for the SCC and Emmy/Grete. The goal of this report is to evaluate the performance viability of unifying both monitoring solutions, replacing SCC's InfluxDB[5] and Telegraf[6] with Prometheus and `node_exporter`. As part of this, the following contributions were made:

- Designing a methodology for benchmarking a pull-based monitoring system.
- Designing a methodology for benchmarking a pull-based monitoring client daemon, both in terms of throughput and the performance degradation caused to typical, throughput-oriented HPC load.
- Benchmarking the performance and scalability of Prometheus for a HPC use case.
- Benchmarking the performance and performance penalty of `node_exporter` for a HPC use case.

1.2.1 Structure

Starting with Section 2, the general topics of monitoring, time series databases and Prometheus get introduced. Related work about time series database performance will be analyzed. After that, in Section 3, the benchmark methodology for both Prometheus itself as well as `node_exporter` will be explained. Then, in Section 4, the results of those benchmarks will be shown. After a short discussion in Section 5, the work will be concluded in Section 6.

2 Background: Monitoring

Monitoring consists of 2 important components. On the one hand, it is the continuous collection of mostly numerical data and metrics from various systems. On the other hand, it is the aggregation and analysis of this collected data/metrics within a certain period of time, up to real-time monitoring.

Several different kinds of metrics can be analyzed using monitoring systems. From typical system metrics such as CPU, RAM, disk, or network usage to monitoring whole server racks with power and cooling statistics as well as application specific metrics such as databases or webservers.

Most monitoring systems follow the collector/database/dashboard architecture. On each node, a *collector* is running as a daemon, exposing the internal metrics for an centralized database, aggregating all nodes. Those databases are usually either general, document-based NoSQL databases such as MongoDB[7] and Elasticsearch[8] or specialized TSDBs such as InfluxDB[9] or Prometheus[10].

[5] `https://www.influxdata.com/products/influxdb-overview/`
[6] `https://www.influxdata.com/time-series-platform/telegraf/`
[7] `https://www.mongodb.com/`
[8] `https://www.elastic.co/elasticsearch`
[9] `https://www.influxdata.com/`
[10] `https://prometheus.io/`

To provide an taxonomy, most monitoring systems can be categorized on two dimensions:

Cloud-based or On-Premise: Monitoring systems can either be managed, cloud-based or self-hosted, on-premise services. Both have their advantages and disadvantages.

Cloud-based monitoring services such as Datadog[11] or Splunk[12] have several advantages. By using an externally hosted service, it simplifies the overall monitoring maintenance. By not requiring hardware for another service, it lowers the barrier of entry. Furthermore, since the whole monitoring stack is written by the same manufacturer, it allows for tighter integration.

On-Premise hostings, on the other hand, also have several advantages over the cloud-based solutions. Since they are part of the local network, they are easier to integrate into current infrastructure, even those parts that one doesn't want to publically expose such as Active Directory or other Identity Management Solutions. Additionally, it mitigates potential security and privacy risks, especially for data that could either pose a security risk such as showing applicable vulnerabilities or data that is legally not allowed to be processed off-premise such as sentitive user data. Lastly, while the software stack is more heterogenous, it allows for more specialization based on the users needs.

Push or Pull: There are two different paragigms in data gathering:

In the more common *push* paradigm, the data-collecting daemon sends the data to an API endpoint. This has multiple advantages: First, it is easier for firewalls, because it does not need any inbound connection establishment. It is also less overhead for the TSDB since it just needs to passively recieve the data. Lastly, it is easier to send non-aggregated raw data, since the client can decide when it has enough data to send to the database.

In the *pull* paradigm, the TSDB itself iterates over, and fetches all metrics, which are exposed as newline-seperated key-value pairs through an specified HTTP-endpoint. The biggest advantage is that the TSDB can't be overwhelmed with requests since it implicitly rate-limits itself through the amounts of outgoing requests. Furthermore, it allows for lazy and just-in-time fetching of data, reducing the overall overhead in the network.

The GWDG uses both push and pull TSDBs in the SCC and Emmy cluster respectively.

2.1 Current Architecture: GWDG SCC

The SCC uses an push-based architecture: On every node, the plugin-powered Telegraf daemon collects all metrics, which are then send to the InfluxDB TSDB. Finally, this database gets queried using the legacy InfluxQL by Grafana[13] as configured in the dashboards.

[11] `https://www.datadoghq.com/`
[12] `https://www.splunk.com/`
[13] `https://grafana.com/`

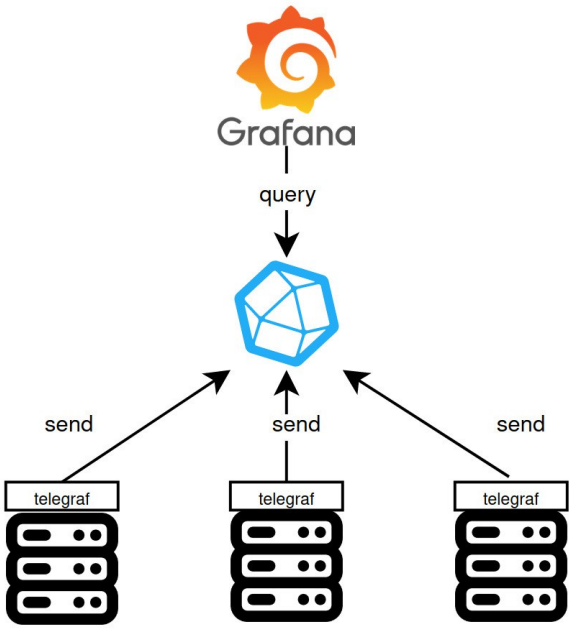


Figure 1: The InfluxDB-based push-setup for the SCC.

2.2 Current Architrecture: HLRN Emmy

For Emmy, a pull-based architecture is used. The node_exporter[14] exposes the `/metrics` HTTP endpoint, to which the Prometheus database connects to when fetching the data. The aggregated data also gets queried through pre-made Grafana dashboards with PromQL.

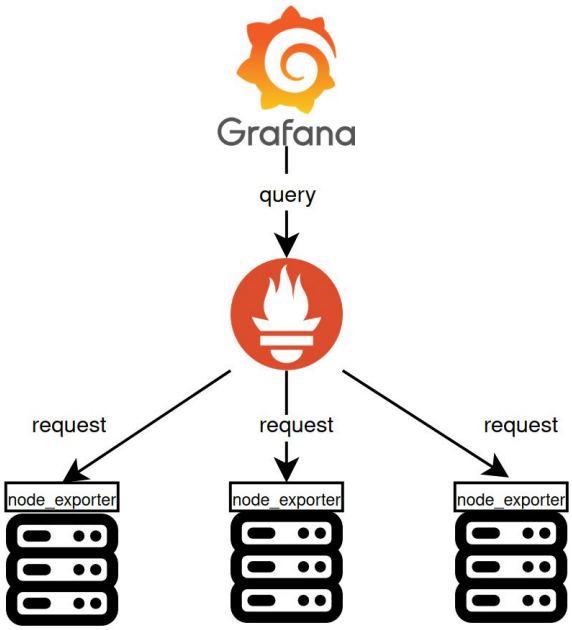


Figure 2: The Prometheus-based pull-setup for Emmy.

[14] `https://github.com/prometheus/node_exporter`

2.3 Prometheus

More than just a TSDB, Prometheus is a pull-based systems monitoring toolkit. Originally developed at SoundCloud, it is now a non-commercial open source project hosted by the Cloud Native Computing Foundation (CNCF), which is a part of the Linux Foundation. The clients expose the pull metrics via `node_exporter` or other custom tooling, that then get scraped by the Prometheus server into its TSDB, which can then be queried through PromQL, its own query language.

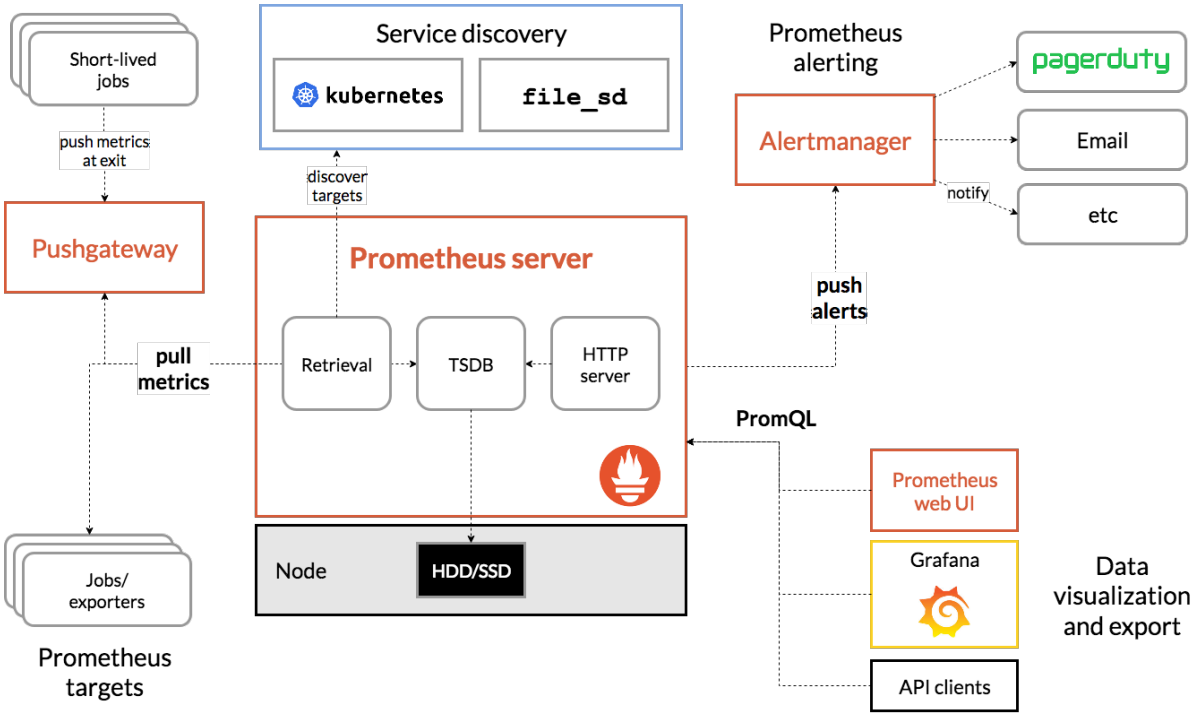


Figure 3: Prometheus high level architecture [5].

2.4 TSDB Performance Evaluation

While there are several papers on TSDB performance evaluation, most benchmarking was mainly done by different vendors. In this section, we will cover one publication as well as two industry standards for push and pull based database benchmarking respectively.

The most actively maintained academic benchmark was first published by Liu and Yuan in 2019 [6]. It is maintained to this day. While it surrently only supports Push-based metrics, it supports not only traditional TSDB such as InfluxDB or TimeScaleDB but also classical SQL-based databases such as SQLite or Microsoft SQL Server. It also supports several benchmarking scenarios testing both write and query performance.

But the canonical benchmarking suite for push-based TSDBs is the Time Series Benchmark Suite (TSBS) [7], maintained by Timescale, the maintainer of timescaledb[15], a time-series database packaged as an PostgresSQL. Initially developed by an external contractor for InfluxData as `influxdb-comparisons` [8], it supports most TSDBs. It is split into 3 different, purely distinct phases: Data and query generation, data insertion, and query execution. This is done to minimize the load generator overhead for more reliable results.

[15] `https://www.timescale.com/`

Due to the constant load configured on the server side instead of the load generator side, pull-based TSDB benchmarking is more difficult. Although still pretty new, the only well maintained benchmark suite for prometheus-like systems is `prometheus-benchmark` [9], developed by VictoriaMetrics[16], a competitor to Prometheus. It supports VictoriaMetrics as well as Prometheus based Grafana's Mimir[17] and the CNCF-maintained Prometheus clustering solutions Cortex[18] and Thanos[19]. Unfortunately, it can not be used for this benchmark as it expects a Kubernetes-based cloud environment while this use case is around HPC usage.

Unfortunately, due to the competitive nature of the highly funded No-SQL startup space, both benchmarks suits are susceptible to commercial incentives.

3 Benchmark Methodology

This section will cover the design and methodology behind all benchmarks. In particular, the following benchmarks were designed as part of this report:

- Measuring the isolated performance of the metric gathering function within the `node_exporter` daemon.
- Measuring the end-to-end performance and stress test resilience of the `node_exporter` daemon with different HTTP load generators.
- Measuring the scalability of Prometheus by increasing the number of daemons to fetch from.
- Measuring the performance penalty of `node_exporter` on an running HPC job (jitter-benchmark).

Note that the code for all benchmarks [10] as well as the patched `node_exporter` [11] can be found on Github.

3.1 Setup

For the `node_exporter` performance benchmarks as well as the Prometheus benchmark, an HPC gcn2 type node from the HLRN Emmy cluster was used. This node has two Xeon Platinum 9242 with 48 cores each as well as 376GB of RAM. The servers were solely used for the benchmarks, thus resulting in no noisy neighbour problems.

In order to facilitate a more minimal operating system with less running services, the jitter-based performance penalty benchmark was done locally on a Thinkpad T14 Gen 1 running a minimal Ubuntu Server 22.04 LTS with all non-essential services killed. The Thinkpad has a quad-core Intel i5-10210U and 16GB of RAM.

The `node_exporter` benchmarks uses a precompiled node exporter of version 1.7.0[20] For the isolated metric gathering benchmark, `node_exporter` was forked from version

[16] `https://victoriametrics.com/`
[17] `https://grafana.com/oss/mimir/`
[18] `https://github.com/cortexproject/cortex`
[19] `https://github.com/thanos-io/thanos/`
[20] With the following collectors enabled: cpu, cpufreq, infiniband, meminfo, netdev, vmstat.

1.7.0. For Prometheus, a Ubuntu 22.04 based singularity container was used as a basis, containing a Prometheus version 2.45.1. The Dockerfile is also available in the repository [10].

3.2 node_exporter

3.2.1 Metric Gathering

When requesting all metrics via the `/metrics` HTTP endpoint, `node_exporter` runs the following function:

```go
// Collect implements the prometheus.Collector interface.
func (n NodeCollector) Collect(ch chan<- prometheus.Metric) {
	wg := sync.WaitGroup{}
	wg.Add(len(n.Collectors))
	for name, c := range n.Collectors {
		go func(name string, c Collector) {
			execute(name, c, ch, n.logger)
			wg.Done()
		}(name, c)
	}
	wg.Wait()
}
```

Listing 1: How the metrics are collected in `collector/collector.go`

So when requesting the metrics, `node_exporter` spawns a new green thread for each metric plugin, awaiting all results in a fork-join like model with a semaphore. Instead of benchmarking the single function by mocking a realistic application state, the Collect function was patched as follows:

```go
// Collect implements the prometheus.Collector interface.
func (n NodeCollector) Collect(ch chan<- prometheus.Metric) {
  [...]
        // BENCHMARK BEGIN
        N := 1000
        // record the time
        totalStart := time.Now()
        for i := 0; i < N; i++ {
                n.RealCollect(ch)
        }
        totalEnd := time.Now()
        fmt.Println("total time: ", totalEnd.Sub(totalStart)
    .Milliseconds())
        fmt.Println("average time: ", float64(totalEnd.Sub(totalStart)
    .Milliseconds())/float64(N))
}

func (n NodeCollector) RealCollect(ch chan<- prometheus.Metric) {
        wg := sync.WaitGroup{}
        wg.Add(len(n.Collectors))
        for name, c := range n.Collectors {
                go func(name string, c Collector) {
                        execute(name, c, ch, n.logger)
                        wg.Done()
                }(name, c)
        }
        wg.Wait()
}
```

Listing 2: The patched collector measuring the collection time. Note that the `RealCollect` function contains the same code as the unpatched `Collect`.

Further small patches had to be done to fix any errors, see the reporitory for more information [11].

3.2.2 End to End

The end-to-end benchmarks measured the throughput of the client exposing the metrics using traditional HTTP benchmarking. In particular, two different benchmarking tools were used:

- **wrk** [12] is a popular, CLI-based HTTP benchmarking tool written in C. Due to its optimized performance it can serve as a baseline optimal throughput. In order to further improve throughput, it keeps all HTTP connections open between requests.
- **go-wrk** [13] is a reimplementation of wrk in the Go programming language, using gos green threads and standard `net` library for the load generation.

All benchmarks are continuously measured using `vmwstat 1`. The benchmarks can be divided into three different kinds.

1. **wrk sequential**: Using a single thread and a single connection, this is the most realistic load, as metrics are usually only crawled by a single Prometheus server as well. This is also the simplest possible benchmark; just send as fast as possible.

2. **wrk parallel**: Scaling along the number of open HTTP connections and threads. This is for creating the maximally possible throughput, although it is not realistic since prometheus does not open multiple HTTP connections to the same `node_exporter`.

3. **go-wrk parallel**: Scaling along the number of go routines/green threads. This is a more realistic load, as it most behaves like Prometheus. Both Prometheus and go-wrk use the same underlying request library, both do not keep HTTP connections open between requests and both use go routines for parallelism.

The benchmarks work as follows: In an outer loop, we scale around the number of processes consumed by the `node_exporter` by changing the `GOMAXPROCS` environment variable of the underlying go runtime. Note that even with single threading go routines can still highly improve performance by hiding I/O idle times between different metrics fetching. For each parallelism, we scale the software as described above, sending against the `/metrics` endpoint, which in turn starts node exporter to refetch and return the newest values. By measuring how often this request can be done and the latency it took, the performance can be evaluated.

Note that this complete benchmarking pipeline is scripted out and automated into a single SLURM job.

3.2.3 Jitter

While the performance of the metric collector itself is important, it can reasonably be assumed that they are able to run well on big HPC machines. From a raw compute providers standpoint, the more interesting question is whether the metric collection overhead is managable, i.e. the performance degradion of the running jobs are significant.

Performance degration is hard to measure. For once, HPC systems, while running a minimal operating system, are still very complex running many different daemons. It is not sufficient to just measure the time a single threaded metric collection takes and take that share times the CPU clock speed; other performance losses such as flushed CPU caches or bad OS scheduling can incur even bigger losses than the actual runtime. As previously shown `node_exporter` spawns a new go routine for each metric, which could result in a lot of cache invalidation.

The approach used in this report is a more realistic one. We have written an MPI-based program running the following pseudo-code:

```c
#define N 1000000
timestamp BIG_ARRAY[N];

int main() {
  MPI_INIT();
  for (int i=0; i<10; ++i) {
    for (int j=0; j<N; ++j) {
      BIG_ARRAY[j] = get_time_now();
    }
    save_big_array();
    MPI_BARRIER(); /* sync */
  }
}
```

Listing 3: A simplified C-style pseudo code of the jitter program.

This code just, as fast as possible, measures the current time on as many processes as spawned via MPI. By using barriers, it can be assured that the processes do not get out of sync, creating unexpected load spikes. Once the `node_exporter` metrics get requested, a slowdown in measurements can be seen, which can then be analyzed in order to quantify the severity of the interference.

3.3 Prometheus

As examplained in the background section, benchmarking pull-based TSDB monitoring solutions is comparitively complicated. Using normal push-based APIs, one can just use traditional HTTP load generators such as JMeter. This is not possible, since the TSDB itself has to start each request to the metric collector daemons. So, instead of measuring how many requests the database can handle, it will instead be measured how many clients Prometheus can pull from until its pull frequency suffers. This is done my completely mocking the `node_exporter` collection daemon and counting the number of times each node was requested by Prometheus.

More specifically, Prometheus gets an autogenerated config[21] that configures it to scrape all targets, i.e. all `node_exporter`, to be checked every 10 seconds. This benchmark runs 10 minutes. This means that, once all clients report to be crawled significantly less than 60 times, Prometheus was not able to handle all mock clients anymore, which implies that the load was too much.

The mock clients were created using Python 3.9 with FastAPI[22] 0.104 and the Uvicorn[23] 0.24 as an Asynchronous Server Gateway Interface (ASGI) implementation. Each node exposes 50 integer metrics, although this is configurable. Instead of just using a static website, each metric updates randomly in order to circumvent any kind of caching optimization on Prometheus side.

For randomness, Python uses a standard mersenne twister with a timestamp as a seed value. Since all mock APIs are spawned at approximately the same time, it could

[21] Example config in Appendix.
[22] `https://fastapi.tiangolo.com/`
[23] `https://www.uvicorn.org/`

happen that multiple nodes have the same mock values. Since it was not known whether Prometheus optimizes/joins multiple nodes with the same values, this had to be fixed.

One could theoretically call `/dev/urandom` for every number, but both the OS call and the Python bytes-to-number conversion are too costly. So instead, when starting the number, the following prodecure is run to insure cryptographical uniqueness between the different mock clients:

- Get 8 random bits from `os.urandom`, interpret as unsigned byte b.
- Let the mersenne twister generate b random numbers as a warmup.
- Use the warmed up mersenne twister as a random number generator.

When terminated[24], each mock client dumps the amount of requests it recieved into a text file. Concurrent writes are fixed by letting each client create its own text with with its current process id interpolated into the filename.

To summarize, the high level fully automatized workflow works as follows:

- The number of mock clients are provided as an CLI parameter in the form of a free port range to use.
- Given that port range, a Prometheus config is dynamically generated, configuring the service to scrape each port once every 10 seconds. That config is later bind-mounted into a custom Prometheus singularity container.
- Spawn all mock clients; sleep $2 \cdot$ `number_of_clients` for them to start.
- Record server utilization in 1 second resolution using `vmstat 1`.
- Start the singularity container with the config and a data directory bind-mounted in.
- Sleep for 10 minutes (the actual benchmark).
- Terminate[25] prometheus, then `vmstat`, then all mock exporter.

4 Results

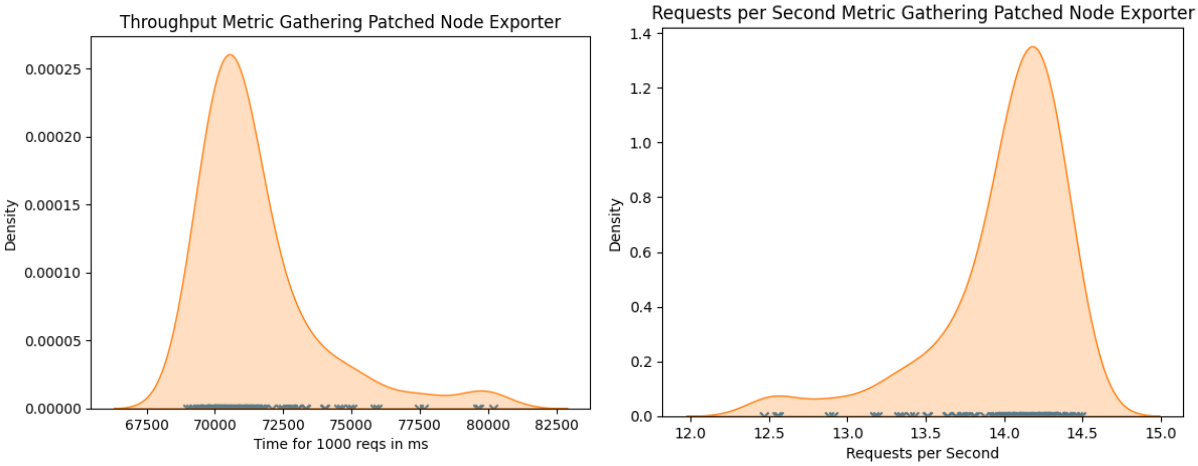
This section will cover the results of the aforementioned benchmarks. First, it will go over the performance of the `node_exporter` itself, which measures how performant the daemon is at collecting all metrics of a single node. After that, the focus will be on the overhead on normal computation load incured by the `node_exporter`. Lastly, the results of the benchmark of Prometheus itself will be covered.

Note that all benchmark scripts as well as analysis code can be found in the accompanying GitHub repository.

[24] This is done via explicit signal handling.
[25] The exact signal is `SIGTERM`.

4.1 `node_exporter` Performance

Patched Node Exporter: First, we start looking at the metric gathering benchmark, i.e. running the metric gathering 1000 times in a loop without the overhead of HTTP and the Go router. The following probability density functions were computed using `seaborn`s kernel density estimation.



(a) (b)

Figure 4: A kernel density estimation of the results. a) showing the raw times it took to collect all metics 1000 times. b) calculated the approximated requests per second from the initial data.

Calculating from the original data, the average request took 71ms, which is an equivalent of 13.99 requests per second, which shows that there is plenty of spare time for adding additional metrics. For the realistic use case of running Prometheus as the monitoring solution of the SCC, it can be assumed that the metrics will be fetched around once every 10 seconds. This results in 0.71% of the time a go process will be running, allocating a single CPU. As previously mentioned, the node used has two Xeon Platinum 9242 with 48 cores, i.e. 96 threads, each. Depending on how well multithreading is utilized at that moment, this results in somewhere between 0.014% and 0.0074% of the total compute time. Although this data shows that the direct resource overhead incured by `node_exporter` is negligable note that this computation does not include the performance penalty on the other jobs and thus can be seen only as a loose lower bound.

Node Exporter HTTP benchmarking: The results of this benchmark are two-fold. First, it was analyzed how well it scales with with parallelism under moderate load. For that, a single threaded `wrk` load generator[26] was used while the number of processes available to the Go runtime was increased between benchmarks:

[26]Using multiple open HTTP connections

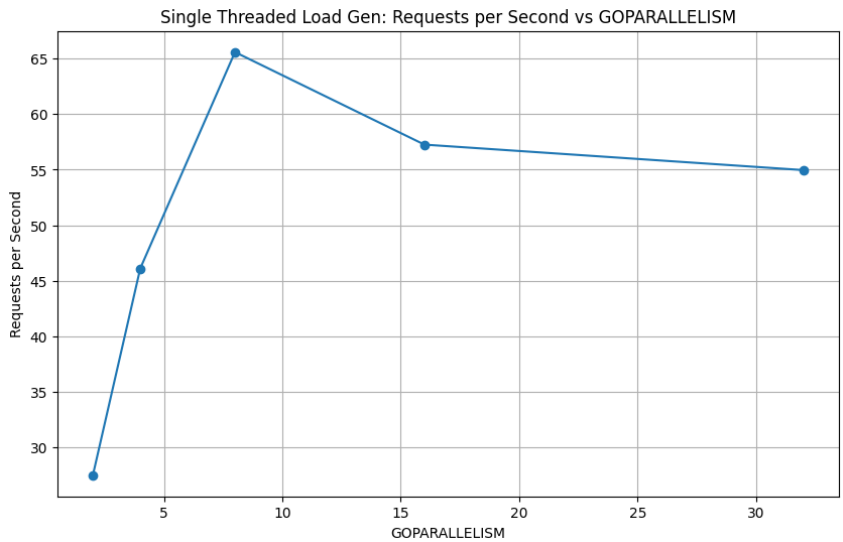
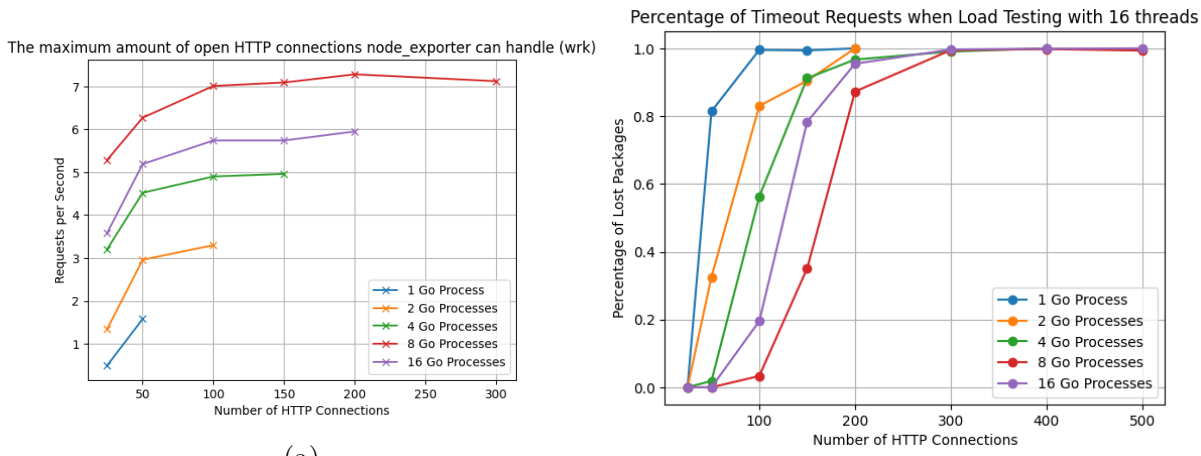


Figure 5: The requests per second given the amount of processes available for Gos runtime. Note that $n = 1$ failed while benchmarking due to an error in the benchmark pipeline.

It can be seen that beyond 8 processes the performance decreases, due to the overhead being higher than the performance gains made my the extra parallelism. Furthermore, with 2 processes, the whole end-to-end request[27] was completed around 27.5 times per second.

Lastly, it was tested how resilient the `node_exporter` is against very heavy load. For this, the `wrk` load generator was used using 16 threads. The benchmark was scaled in two dimensions: Both in the number of concurrent HTTP connections between `wrk` and `node_exporter` as well as the number of processes available to the Go runtime.



(a) (b)

Figure 6: a) shows the average throughput and b) the percentage of requests which timed out.

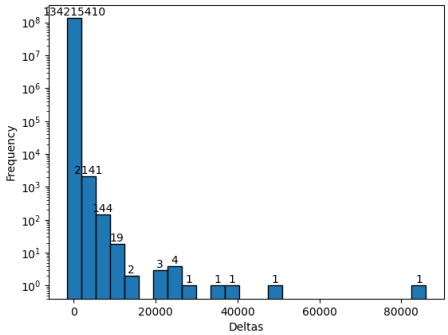
This benchmark shows that `node_exporter` performs very poorly under heavy load. Even with the previously found optimum of 8 OS processes, it results in over 80% of

[27] Excluding a realistic round trip time since the request was done against `localhost`

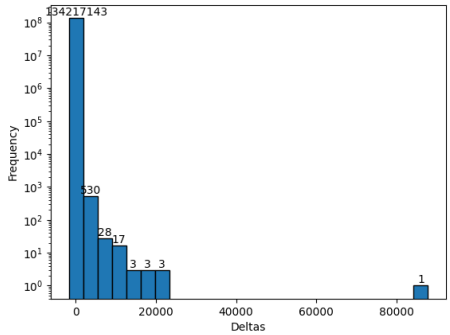
requests timed out when sending under a heavy load.

4.2 `node_exporter` Performance Penalty (Jitter)

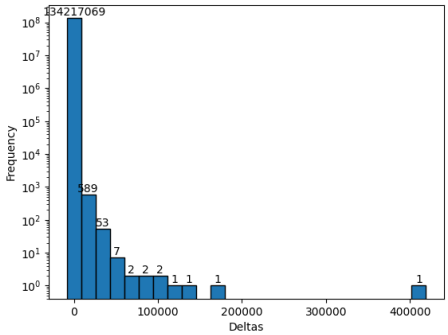
The following histograms are measurements from the jitter benchmark ran on an 8 thread CPU, resulting in $n = 134,217,728$ data points per plot.



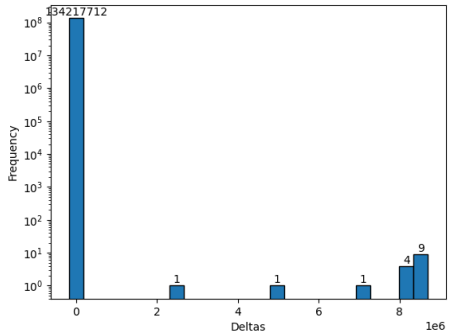
(a) jitter single thread
$t = 2.437996071$



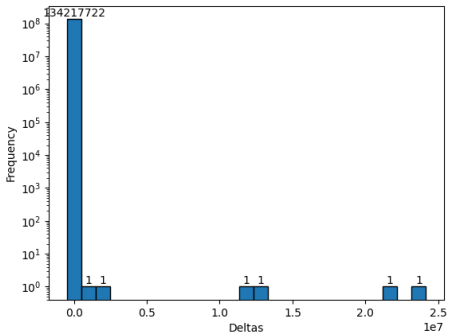
(b) baseline single thread
$t = 2.421950996$



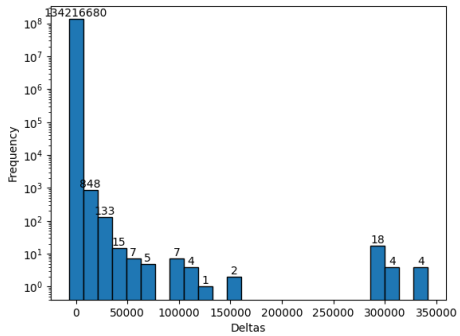
(c) jitter 8 MPI processes (rank 0)
$t = 5.836635677$



(d) baseline 8 MPI processes (rank 0)
$t = 5.968865865$



(e) jitter 8 MPI processes (rank 1)
$t = 5.914636448$



(f) baseline 8 MPI processes (rank 1)
$t = 5.864066244$

Figure 7: The results for single nodes, plotted onto a 25 bin, linearly sized histogram.

Due to the enormous size of the data set, a more sophisticated analysis was not possible; an in-depth reasoning is provided in the Discussion below. But it can be seen

that, while a clear trend of a longer tail can be recognized, it can also be seen that this tail is very thin containing only a few data points. This implies that the performance penalty is incured just a small share of the computation time.

All pre-computed histograms for all configurations and all MPI ranks[28] as well as the fully automated benchmarking workflow and a verbose description are available in the accompanying GitHub repository.

4.3 Prometheus

For this benchmark, the Prometheus was configured to request each node once every 10 seconds for 10 minutes, i.e. to request each node a total of 60 times. From this maximum the theoretical max of 60 times number of nodes can be calculated. All mock clients as well as prometheus ran on the same node.

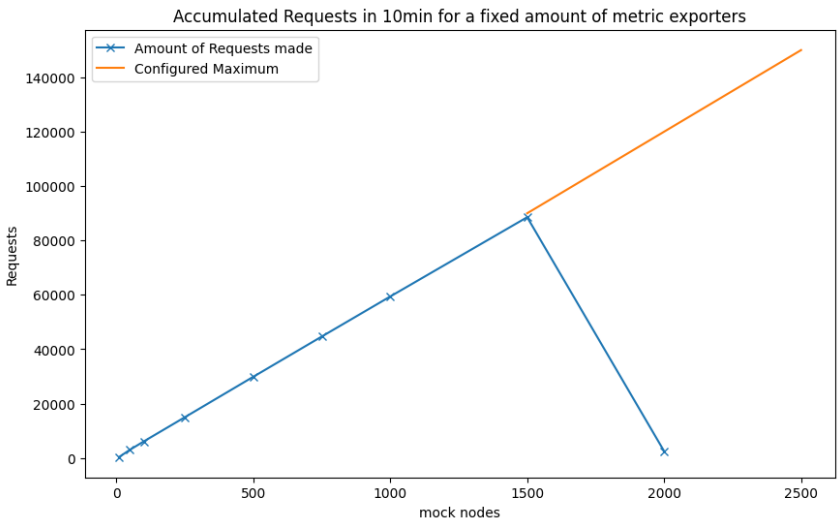


Figure 8: Scaling up the number of mock nodes. The number of requests are the number of times Prometheus contacted one of the mock clients. All mock clients ran on a single node.

The drop off will be further analyzed in the discussion. To ensure that it is not a limitation of the Linux kernel having too many open file descriptors and running processes, the mock clients were then split onto two nodes.

[28] With a bucket size of 10, 25, 50, 100 as well as calculated using the Square-root choice, Sturges Formula and Rice Rule. Also with the tail 0.05% cut off.

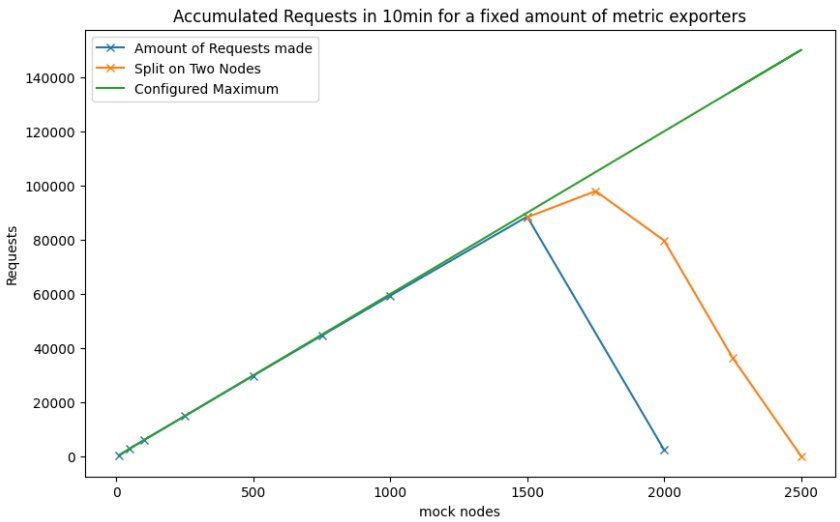


Figure 9: Scaling up the number of mock nodes. The number of requests are the number of times Prometheus contacted one of the mock clients. The mock clients were split onto two nodes.

As one can see, using a single node, the performance drops off sharply after 1500 nodes. And even with the load split onto two nodes, it drops to nearly no requests at all using 2500 nodes.

Excluding the dropoff itself, this performance is very bad. The data implies that, using our single node setup, monitoring more than 1500 clients would fail. Looking at the optimum case of a single node setup fetching 1500 node exporter, it approximately matches 90000 requests, which results in 150 requests per second. The metrics were of the format `keyname3: 42`.

Assuming that each number is in the normal 32bit integer range, each metric can be rounded up to $10 + \lceil \log_{10}(2^{31}) \rceil = 20$ byte of data, with 50 metrics this adds up to 1kb per node. Thus a total throughput of only around 150 kb of useful data per second is achived.

5 Discussion

As clearly seen in the data, the Prometheus job had a scaling problem when moving from 1500 to 2000 mock node exporter. Confusingly, the problem was not CPU, Memory or I/O utilization, as all of those were not fully utilized. The data showing the utilization while benchmarking with one server running 2000 mock nodes can be found in the appendix.

The working hypothesis was that the Linux kernel had problems with that many mock processors; either handling all the open file descriptors or just scheduling the sheer amount of processes. To validate that hypothesis, the two HPC node benchmarks were done. While the benchmark show that it scales further, it does not show that the problem was with either the amount of processes or the amount of file descriptors. When splitting the mock load generator onto two nodes, the expected result would be around twice the amount of nodes until it stops serving all collectors, which is definitely not the case[29].

[29] Not exactly twice the amount, because one of the nodes still ran Prometheus. Although this should not be a problem since, as already mentioned, the node was idling on all ressource dimensions.

Furthermore, one should note that the mocked `node_exporter` used pseudorandomness[30] for the fake metric generation. While this is less realistic than using more sophisticated data generation methods such as perlin noise, it incurs significantly less overhead per fake node. Other benchmarks such as the previously mentioned TSBS used pregenerated mock data. This would also not be possible as it would result into a very heavy I/O load on the central storage cluster, resulting in strongly reduced I/O speed for all Prometheus operations.

Lastly, the jitter benchmark was very difficult to analyze. Due to the inherent design on recording as many data points as possible, it resulted in over 130 million data points, sometimes recording less than 3 seconds of time. It was not possible to downsample the data since the thin long tails were explicitly of interest. The data set was too large to use kernel density estimations to create a probability density function. Furthermore, due to the sheer differences in tail latencies, layering the measurements and its baseline on the same histogram was not possible, as it was not possible to find a bin selection that properly presented both.

It was not possible to properly analyze the tail, because it was very hard to isolate. While for example with the 1 thread benchmark the tails measurements went up to the 1e6 nanoseconds range, the $p_{99.9}$ was only 786ns. The thinness and length of the tail latencies, its range difference between the benchmarks combined with the sheer data set size made any sophisticated analysis impossible within the scope of this report. In fact, this analysis was only possible because the plots could be computed on a HPC node; On the laptop the benchmarks were made of the process was killed, most likely due to an Python-internal out of heap memory error.

6 Conclusion

For this report, four methods of benchmarking the Prometheus monitoring architecture were designed and implemented, measuring both the performance of as well as performance penalty incurred by the `node_exporter` metric collector as well as the performance and scalability of Prometheus.

The results show that, although the collector did not handle the stress test load well, its throughput and overall performance is cufficient and the collection time acceptable, even using only a single process for execution. Prometheus itself did not scale well beyond a certain amount of nodes, failing with 2000 collectors when all run on one not fully utilized HPC node. Thus, a non clustered Prometheus solution would not be able to serve the combined load of both the SCC and Emmy cluster as Prometheus performance degrades too much immensely after a certain amount of nodes.

6.1 Future Work

While a simple Prometheus showed to not scale properly enough, clustered Prometheus-based solutions such as Cortex[31] or Thanos[32] could provide a solution able to monitor

[30] Using the warmed-up mersenne twister method mentioned in the methodology.
[31] `https://cortexmetrics.io/`
[32] `https://thanos.io/`

multiple HPC clusters[33]. Possible future work includes evaluating and comparing both of these clusters.

Furthermore, in order to have a definite conclusion on the full performance penalty created by `node_exporter`, a more sophisticated analysis has to be done. This could include identifying the tail through clustering techniques, manually cutting of the few very extreme data points for more insightful plotting or otherwise computing a cutoff for the data points.

Lastly, the Prometheus benchmark could be set up in a way that Prometheus itself is deployed isolated on a single node while all mocked collectors are distributed on different nodes. This would create a more realistic scenario with higher round trip times between all nodes.

[33]The collector itself had sufficient performance.

References

[1] Vitalii Shevchuk. *GPT-4 Parameters Explained: Everything You Need to Know*. Medium. July 17, 2023. URL: https://levelup.gitconnected.com/gpt-4-parameters-explained-everything-you-need-to-know-e210c20576ca (visited on 12/08/2023).

[2] *Scientific Compute Cluster*. Section: hpc. URL: https://gwdg.de/en/hpc/systems/scc/ (visited on 12/08/2023).

[3] *HLRN-IV-System Emmy*. Section: hpc. URL: https://gwdg.de/en/hpc/systems/emmy/ (visited on 12/08/2023).

[4] *HPC DLR CARO*. Section: hpc. URL: https://gwdg.de/en/hpc/systems/caro/ (visited on 12/08/2023).

[5] Prometheus. *Overview | Prometheus*. URL: https://prometheus.io/docs/introduction/overview/ (visited on 12/15/2023).

[6] Rui Liu and Jun Yuan. *Benchmarking Time Series Databases with IoTDB-Benchmark for IoT Scenarios*. Apr. 18, 2019. arXiv: 1901.08304[cs]. URL: http://arxiv.org/abs/1901.08304 (visited on 12/15/2023).

[7] *timescale/tsbs*. original-date: 2018-08-08T14:30:28Z. Dec. 15, 2023. URL: https://github.com/timescale/tsbs (visited on 12/15/2023).

[8] *influxdata/influxdb-comparisons*. original-date: 2016-03-25T20:27:10Z. Nov. 23, 2023. URL: https://github.com/influxdata/influxdb-comparisons (visited on 12/15/2023).

[9] *VictoriaMetrics/prometheus-benchmark*. original-date: 2021-12-09T11:58:40Z. Dec. 11, 2023. URL: https://github.com/VictoriaMetrics/prometheus-benchmark (visited on 12/15/2023).

[10] Lars Quentin. *lquenti/tsdb_comparison*. GitHub. URL: https://github.com/lquenti/prometheus_evaluation_hpc (visited on 12/15/2023).

[11] Lars Quentin. *lquenti/node_exporter*. original-date: 2023-11-14T07:50:08Z. Nov. 14, 2023. URL: https://github.com/lquenti/node_exporter (visited on 12/15/2023).

[12] Will Glozer. *wg/wrk*. original-date: 2012-03-20T11:12:28Z. Dec. 15, 2023. URL: https://github.com/wg/wrk (visited on 12/15/2023).

[13] *tsliwowicz/go-wrk: go-wrk - a HTTP benchmarking tool based in spirit on the excellent wrk tool (https://github.com/wg/wrk)*. URL: https://github.com/tsliwowicz/go-wrk (visited on 12/15/2023).

A Example Autogenerated Prometheus Config

```yaml
global:
  scrape_interval: "10s"
  evaluation_interval: "10s"

scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets: [
          "localhost:9200",
          "localhost:9201",
          ...
      ]
```

Listing 4: Example Autogenerated Prometheus Config (target list truncated and reformatted)

B Prometheus Node Utilization data

All data was collected by running `vmstat` each second while running the benchmarks

B.1 I/O was not the bottleneck

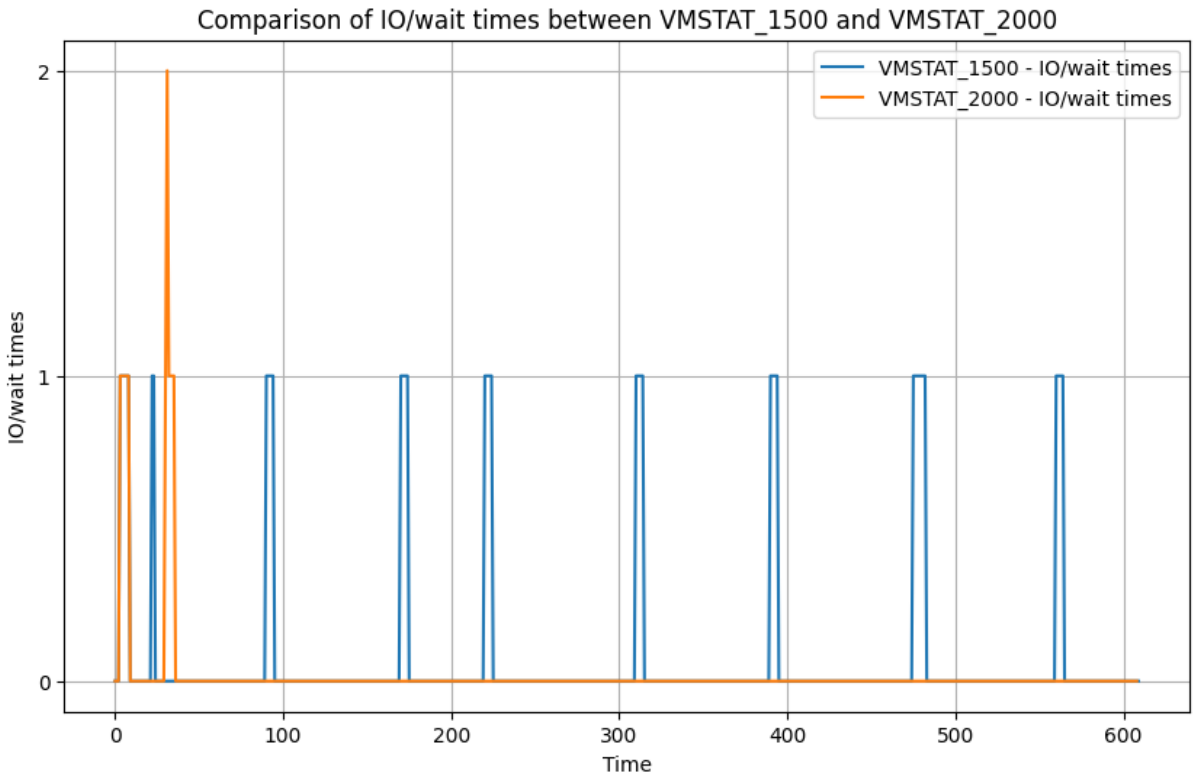


Figure 10: The I/O wait times recorded, showing that there was no increasing queue of I/O operations.

B.2 Memory was not the bottleneck

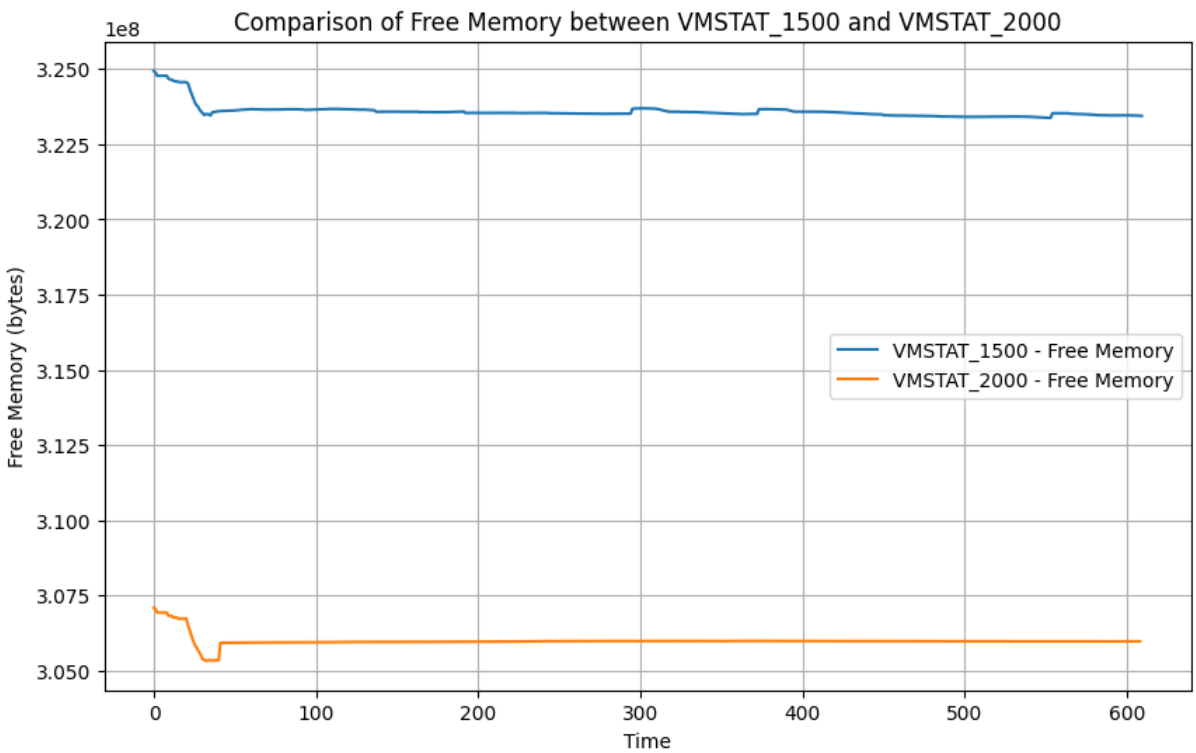


Figure 11: The free memory, showing that there was more than enough free memory, and that the memory usage was not significantly different between the benchmarks.

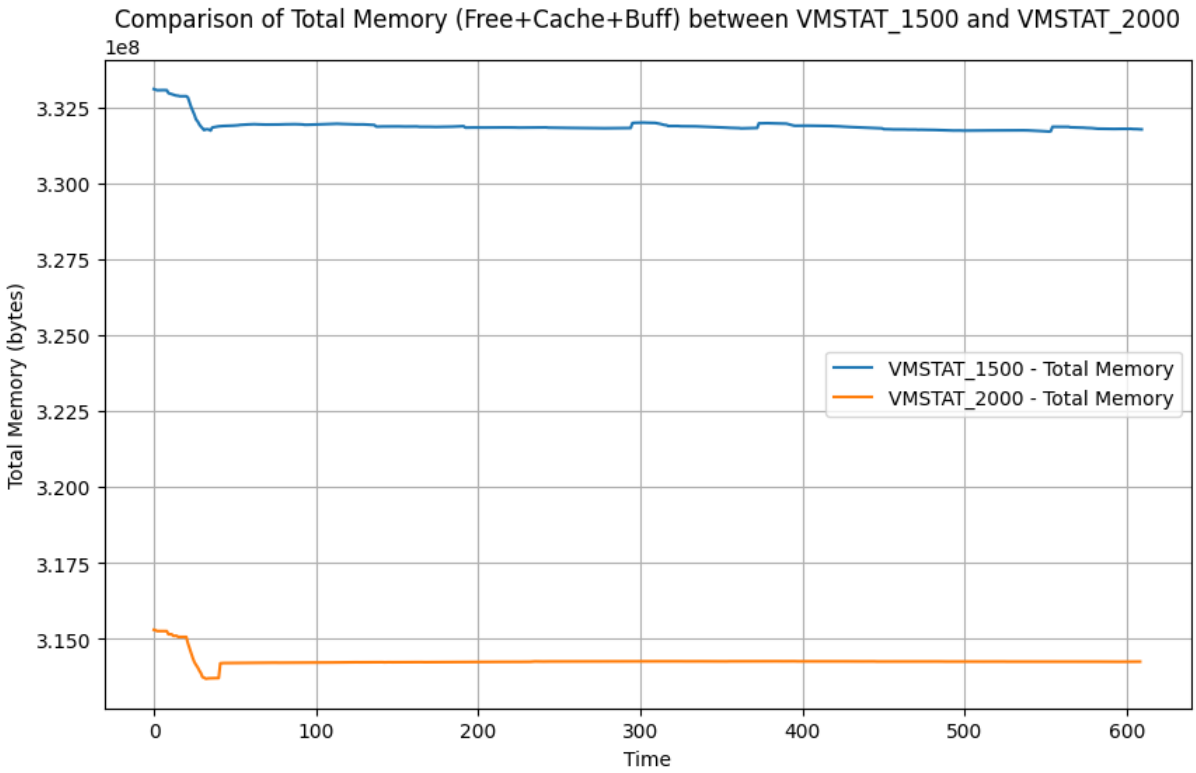


Figure 12: The total non-used memory (including page cache and buffer cache), again showing no significant difference between the well performing 1500 node benchmark and the slow 2000 node benchmark

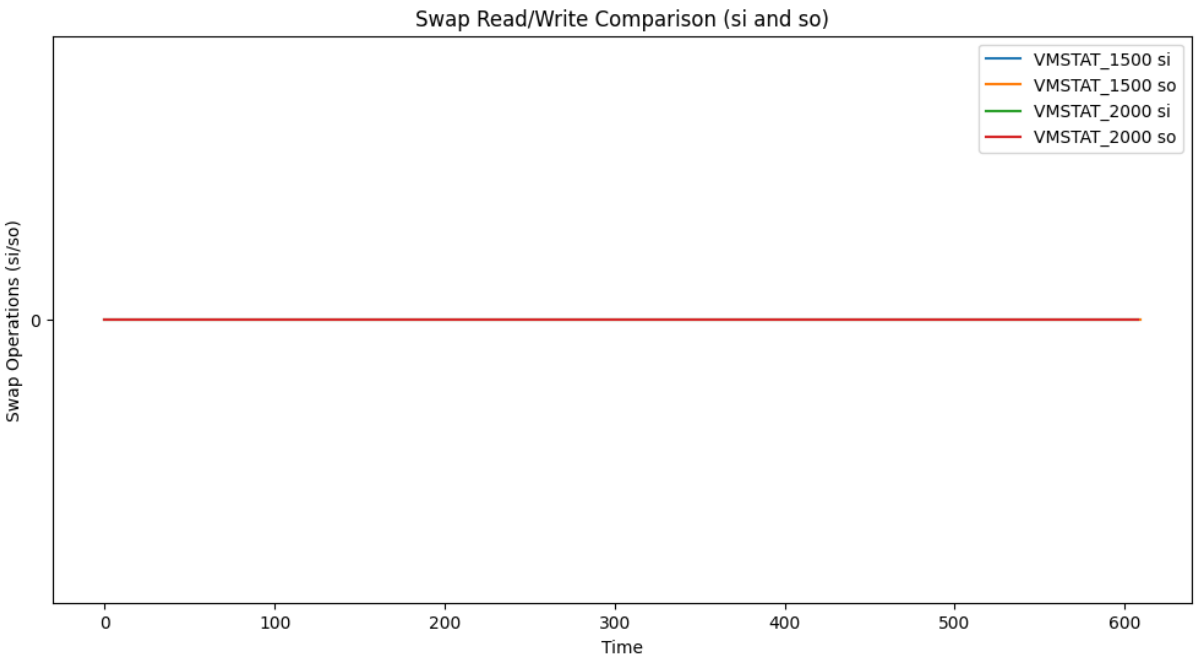


Figure 13: No swap was used at both benchmarks.

B.3 CPU was not the bottleneck

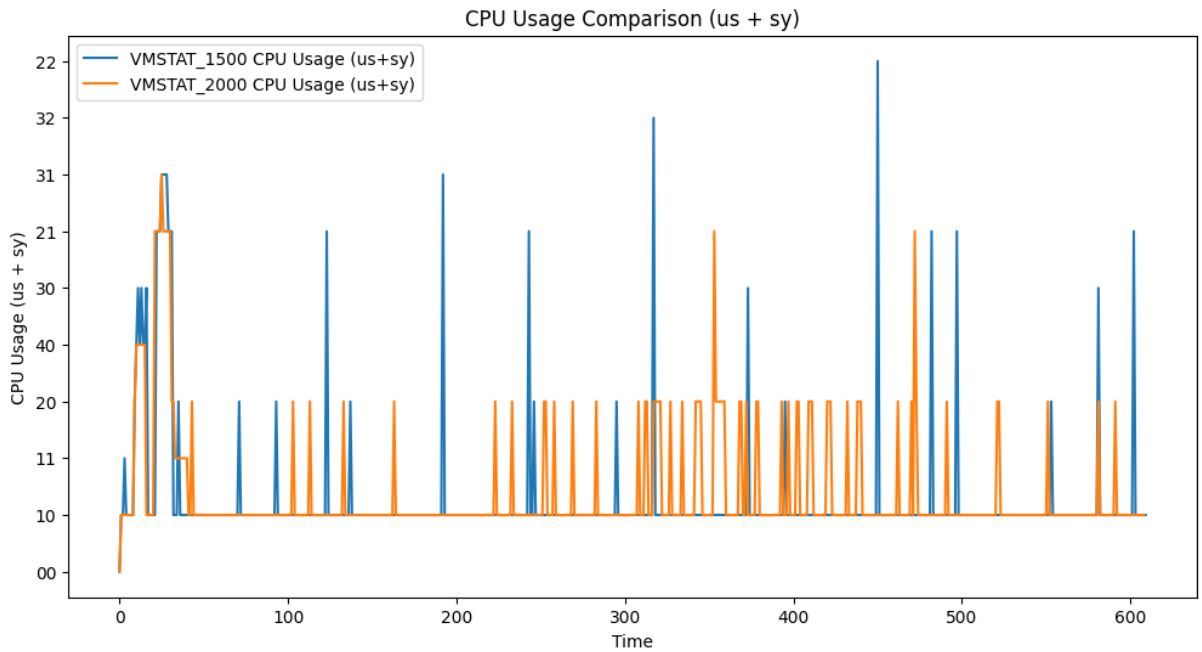


Figure 14: The accumulated CPU usage, both system time (sy) and user time (us), showing that the processor was not fully utilized in both benchmarks, and that due to the high amount of threads available the CPU usage was not significantly different.