Toward the Mixing of Monitoring and Profiling Operations on HPC Platforms

Ilya Meignan–Masson

Toward the Mixing of Monitoring and Profiling Operations on HPC Platforms

Ilya Meignan--Masson
TER intern in the Datamove team
Univ Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
Supervised by: Olivier Richard, Datamove

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature: Ilya Meignan--Masson, 10/06/2022

Abstract

System and job monitoring are two established way of measuring the utilization of HPC systems. Due to the scale and complexity of modern HPC systems, users also require to profile their application using the actual system. However, the usage of many tools usually requires more effort in support and can also lead to significant under-performance. We introduce Colmet, an existing system modified to combine monitoring and profiling operations. Users can dynamically reconfigure Colmet at runtime and set a different sampling period for each metric. We present the results of the performance analysis conducted.

1 Introduction

Modern HPC systems are composed of hundred of thousands if not millions of cores linked to nodes dedicated to storage with a very efficient network. They require large amounts of money and effort for their design, their building and for their operation. At this scale, their efficient usage is an essential goal.

The scale of those system makes tedious any manual monitoring and motivates dedicated monitoring systems. Some systems focus on presenting to system administrators an overview of the status of components in the system (Nagios [13]). In this paper, we define monitoring as the process of collecting, gathering and storing data about the execution of one or several applications. Some systems also include some way of displaying and presenting the collected data to the user. This data can be drawn from various sources and will be called metrics. Metric sources include hardware counters (*e.g.*, using `perf` in Linux systems), operating system performance data (virtual memory or kernel related metrics) or network performance data. In the recent years, the issue of energy consumption has become a crucial matter as the biggest systems now require tremendous amounts of energy to operate, for their nodes and their cooling [1]. Metrics are collected periodically. We call sampling period the interval between two moments in time where the system collects data.

Designing monitoring systems for HPC platforms has the following challenges. First and foremost, the monitoring system will interfere with user applications running on the nodes. To avoid tampering as much as possible the data, the overhead of the system needs to be as small as possible. Another issue is the complexity of the hardware and software architecture in the system. Monitoring systems designers must choose between two directions. Either supporting and optimizing a specific architecture, which often results in a custom monitoring tool for each system. Or building a portable system which might not suit perfectly the monitored platform but in return will not require much effort to use in many places. Once the data is collected, another issue is to present it in a usable and meaningful way.

In a cluster, users are allocated only a subset of the nodes in what is called a job. In the development process, users of the HPC systems might find the need to perform a profiling of their job. This set of techniques, that aims at optimizing the performance of an application, is crucial in the context of HPC. Due to the complexity of the HPC system, executing the application on another system will only yield limited results. To answer this specific need, distributed application profiling tools have been developed like the ones we present in Section 2. Designers of such tools face the same challenges that we described for monitoring systems. Moreover, a system where every user would be running its own profiling tool next to the global monitoring system would most likely suffer significant under-performance.

In this article, we propose to use the same tool for the system administrators and their monitoring needs as well as for the users and their profiling needs. We implemented this in Colmet [2], an existing monitoring system previously developed in the Datamove team. We modified the tool so that it could collect data at different sampling period for each metric. Our system limit the performance overhead both on the compute nodes and on the network with simple string substitution. It is also rather portable with ZeroMQ sockets for communication which is very common but also cgroups v1 which requires a specific set of Linux kernels to be running on the compute nodes. We believe our system will simplify the usage for both system administrators and users. Users will be provided a integrated and optimized way of profiling their application and system administrators will only have this unique tool to maintain and to fit to the system.

The paper is structured as follows. Section 2 gives an overview of multiple distributed monitoring and profiling systems. Section 3 details the design and the architecture of our system. The performance analysis that we conducted is presented in section 4 before the discussion about limitations and future work in section 5.

2 Related work

Monitoring tools are often customized to the system they are supporting for performance reasons if not built for it. However, modern systems have some usual properties that we describe in this section. We also present some profiling software designed for distributed systems.

To begin with, we note the existence of a recent review of different monitoring systems for HPC systems [20]. This review identifies one of the problems of monitoring large scale HPC system as requiring multiple tools with "custom scripts to get comprehensive monitoring of the HPC system" (section 3.1). It also mention that "there is no single tool to achieve all aspects of monitoring, analysis and alerting" (also section 3.1). On this part, the work presented here addresses part of those aspects while adding a profiling use-case.

The type of metrics that a system can collect is one the most important properties. Some systems will collect CPU performance counters values, virtual memory statistics and kernel related metrics (LIKWID [17], DCDB [16], Colmet [3]), I/O or filesystem related metrics (Beacon [21], Colmet, DCDB), temperature (ExaMon [7], Colmet, DCDB) or Network performance (Colmet, DCDB). Some metrics are collected *in-band* (*i.e.* on the compute nodes) and some must be accessed *out-of-band* (ie. on service nodes or on the routers).

A desirable property for a monitoring system is the ability to store data as a time series. As their name suggest, time series are collection of measures performed at successive moments in time. They are particularly important to give informations on the changes of a metric over time. Most modern systems (*e.g.*, NWPerf [15], LIKWID, Colmet) are capable of storing metrics as time series often using dedicated databases.

These time series can contain the collected data at different scopes.

- The node-scope gives insights on the performance of the compute nodes. This level is particularly interesting for the system administrators to keep track of the performance of the nodes over a period of time or to correctly scale the system to the needs of the users. Systems like Ganglia[14] or DCDB are examples with this scope.
- Consequently, job-level scope combines data collected on the compute nodes and job data given by the Resource and Job Management System (RJMS) to store the data corresponding to each job. This enables HPC system users to obtain insights on the performance of their application over time. NWPerf and Colmet are examples of system at this scope. Data gathered with this scope is sometimes used to perform analysis on the job performance and help the non-expert user to understand the data. LIKWID is an example of such tool providing simple analysis to the users. For this scope, a link between the data collected on a compute node and the job assignation data must be made. Collection agent on the compute nodes can be given this responsibility like in Colmet. Other systems like LIKWID rely on the routers to tag the information as it passes by them.

Even though such analysis can be helpful, HPC users are most of the time more knowledgeable about their application than the monitoring system designers. With an API to dynamically change the metrics collected and the sampling frequency at runtime, such users could tune the monitoring along their needs all along the execution of the application. To our knowledge, no system implements this feature.

Due to the volume of metrics, especially when the number of nodes in the cluster or the sampling frequency increases, some systems like Beacon or Colmet compress the data on the compute node before sending it on the network. This compression is a trade-off between overhead on the network and overhead on the compute node. Even if it does not compress the data, NWPerf designers included a network contention analysis in [15].

The criteria presented here are summarized in Table 1.

Profiling tools usually instrument the code or the executable of an application to trigger data collection at specific points of the execution or rather attach to the process and collect both the metric values as well as the instruction that the process is executing. Parallel profiling tools like TAU [19] follow the former principle while other tools like STAT [6] use the latter. They share with monitoring systems some of their properties like the way metrics are collected (*in-band* or *out-of-band*), the time series storing, the per-job profiling.

3 Colmet : Design and Architecture

In this section, we present the design of Colmet with our modifications. Colmet was first designed after the publication of [11] in 2013. After a first version in Perl, a version has been written in Python [3] that is available in production on Grid5000 clusters. To improve the performance, a Rust re-writing had been started but was left unfinished. To mix operations of monitoring and of profiling, the tool needs to collect metrics at a small sampling period. It motivated the refactoring and improvement of the code to leverage the speed of Rust.

In terms of design, our version is close to the Python one on a lot of points. First and foremost, the tool uses some of the same backends (*i.e.*, an abstraction of the protocol or algorithm needed to actually perform the data capture). Backends related to cgroups ([12]) metrics and to `perf` events are implemented but we intend to add the others like temperature, energy consumption (with the Running Average Power Limit - RAPL [9] metrics), or Infiniband related metrics. We note here that the Python version was using the taskstats interface, which is a kernel interface to access data about a process and we changed it to leverage cgroups created by the RJMS. It also reuses Elasticsearch as a time series database. The scope of the monitoring is at the job-level. It relies here on the RJMS to creates a cgroup for the job on the allocated compute nodes.

	Ganglia	NWPerf	ExaMon	Beacon	LIKWID	DCDB	Colmet	Colmet (Rust version)
Metrics	VMstats, OS and custom	HC/`perf` and VMstats	HC/`perf` and Energy	IO	HC/`perf`, VMstats, network and IO	VMstats, OS, IO, and energy	HC/`perf`, VMstats, network, energy and IO	HC/`perf` and VMstats
Scope	node	job	node	node	job	node	job	job
Dynamic reconfiguration								yes
Features for the users				Integrated analysis and visualization	Integrated online and offline analysis		Visualization GUI	profiling use-case
Performance considerations	Hierarchical collection	performance analysis		Compression		performance analysis	ID substitution	ID substitution

Table 1: Summary of the state of the art

Blank means that there is no information on the related paper.
Metrics :
HC : Hardware Counters, OS : proc and sys virtual filesystem related, VMstats : virtual memory related, IO : filesystem related, network : high-performance network related

Colmet is composed of two components : a node agent and a collector. Figure 1 presents the architecture that we describe in the following.The node agent runs on every node in the system. It stores the list of metrics to gather on this node as a tuple containing the metric, the job id and the sampling period. This relation allows to collect data at different sampling frequency for each metric and to link the data with the jobs. This means that some metrics can be set by system administrators on all the jobs for monitoring with a low sampling frequency. Typical values are at the order of the second. While other metrics can be set by the users for their job with possibly a higher sampling frequency, especially under the second. To this end, users can interact with a script to update the metrics to collect or the default sampling frequency on the compute nodes. Each time some metrics should be collected, the node agent queries the corresponding collection backends. The cgroup backend read the data respectively in the files `cpu.stat` and `memory.stat` created in the virtual directory of the cgroup. For perf event data, they are also read from a file created by a call to `perf_event_open()`. Data is then processed and sent to the collector. To avoid using too much the CPU, the processing simply substitutes an id in place of the metric name, sending shorter messages over the network. Unlike the node agent, the node collector is written in Python as we believe that the overhead on the collector node is not a significant performance issue. The collector listen on a port, receives the data and place it in the Elasticsearch database. This means that it requires a server to run. Communication between the node agents and the collector is done with ZeroMQ (ØMQ) sockets.

4 Performance analysis

An important part of this work was dedicated to the measurement of the overhead caused by the monitoring and profiling system. We focused on the execution time overhead. We studied the effects of the following parameters : the number of metrics, the sampling period and the version of Colmet (including a special value without Colmet launched) using 2 benchmarks from the NASA Parallel Benchmarks (NPB) suite as applications. In this section, we describe our experimental set-up, present the results we obtained and the analysis that guided our exploration.

4.1 Experiment setup

The first benchmark is the Embarrassingly Parallel (EP) benchmark. This benchmark accumulates statistics from pseudo-randomly generated numbers. It is compute-intensive with almost no communication and thus provides an estimate of the upper achievable limits for floating-point performance. The second one is the LU Simulated Computational Fluids Dynamic (CFD) application (LU) benchmark. It is intended to accurately represent the principal computational and data movement requirements of a real CFD application and thus is closer to a real application than the previous one (cf. the specification of the NPB [18], in Sections 3.1.1 and 3.2.1). We used the MPI version of both benchmark. The size of the benchmark is quantified by the class. For each number of node, we chose a class that would be long enough to lower the influence of noise in the result.

All our experiments were carried out in Grid5000 [8], more precisely in the dahu cluster of the Grenoble site. Each node

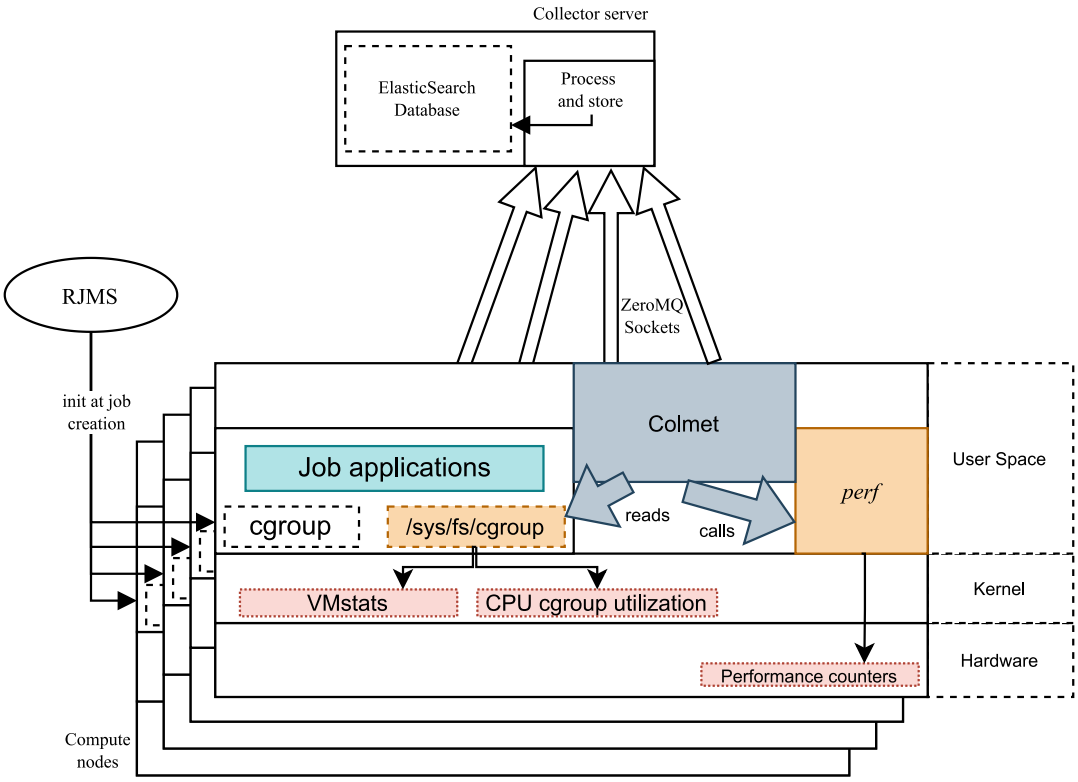


Figure 1: Architecture of Colmet

in this cluster is provisioned with 2 Intel Xeon Gold 6130 CPUs with 16 cores each, 192GiB of memory and a 240GiB of SSD (Samsung MZ7KM240HMHQ0D3). All the nodes are also connected using a simple TCP network as well as a 100 Gbps Omni-Path network. Nodes are running Debian 11 with OpenMPI 4.1.0. Experiments were run using all the cores of the compute nodes but not all the hyperthreads because it would not improve the performance as HPC application computations mainly involve floating point operations and there is usually one Floating Point Unit (FPU) per core.

One technical point worth noting is the usage of the Nix package manager during the experiments to install the required software [10]. This package manager provides strong guarantees on reproducibility. A package is specified by a definition containing the name and the version of all its dependencies. Nix then installs each package in its own environment ensuring that it can run properly as it was designed and tested. A system of linking helps installing each package only once to prevent from using more space than required on the disk.

Usual monitoring systems collect metrics with sampling periods in the order of second. For our use-case of profiling, the monitoring tool should be able to collect metris with a sampling period under the second while keeping the overhead as low as possible. We chose values for the sampling period between usual values like 5 seconds up to the millisecond because we believe that smaller values would not be of interest to the users.

We call configuration a set of values for the parameters. The experimental script [4] requests a job, install the required softwares and executes a set of configuration specified in a YAML file. Because a single experiment can require multiple version of Colmet, an experiment requires multiple run of the script, one for each version. To obtain results as accurate as possible, each configuration is replicated 10 times. We also randomize the order in which the configurations are executed to minimize the observational error. With the 10 values for each configuration, we compute the mean and the confidence intervals with a coefficient of 95%.

4.2 Experimental results and analysis

We present the result of 2 experiment that we performed to measure the impact of Colmet on the performance of the system.

The first one quantifies the impact of the number of metrics collected by the node agent. We ran the LU (class D) and the EP (class E) benchmark with a sampling period of 0.001, 0.01, 0.1, 1 and 5 seconds on 8 nodes (+1 for the collector). We then compared three version of Colmet : the Rust version with only 1 metric, the Rust version with 67 metrics (*i.e.*, the greatest amount the system can gather) and the reference without Colmet. An ideal system would not be impacted by the number of metrics. Results are presented in Fig 2a on the upper row.

The second experiment compares the Rust version with the Python version. To do so, we used the same values as the first experiment and ran them with the Rust version with 67 metrics, the Python version and without Colmet as a reference. Here, our hypothesis is that the Rust version should be better than the Python version but worse than the reference. Results are presented in Fig 2a on the lower row.

The results for the first experiment clearly show that the number of metrics do not have a significant impact on the execution time of the application. The only configuration where the confidence interval are not intersecting is with the LU benchmark at 0.001 second in sampling period.

For the second experiment we do not have, at the moment, enough results to conclude on the hypothesis. Similarly to the first experiment, the execution time with the EP benchmark are spread on a large interval which leads to larger confidence intervals. An hypothesis for this behavior could be that the benchmark is compute-intensive and that Colmet then could be affected to a different core each time leading to non-uniform disruption of the execution which causes such difference between time of the same configuration. On the other hand, the results collected from the experiment with the LU benchmark correspond to the hypothesis. We also observe that, at 1 and 5 seconds, the Python version becomes significantly better than the Rust version. For the moment, we are not able to explain this part.

5 Future work

5.1 Limitations

Considering the problem, namely mixing monitoring and profiling in the same tool, the biggest limitation that this tool exhibits is the impossibility to instrument the code of the application. While this tool is perfectly capable of collecting metrics at a very high sampling frequency, correlating the gathered data with the instructions of the application will be hard if not impossible.

Even if we implemented the mechanism of dynamically changing the configuration of the node agent, it is not production ready especially in terms of permissions. The lack of proper permission system allows users to change the configuration of the node completely.

Another limitation on the design is the scalability. All our experiments were using a relatively small number of nodes with regard to the usual size of small to middle size clusters. At a larger scale, our choice of simple compression might not be enough to prevent an important overhead on the network because all the node agents are sending to the same collector.

5.2 Directions

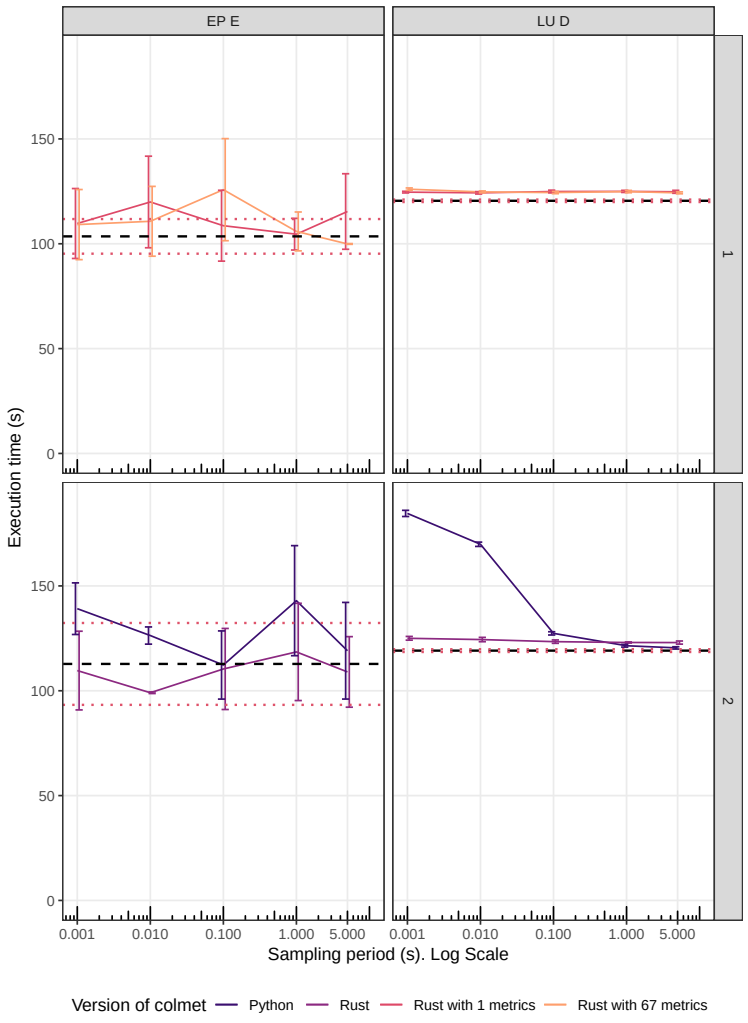
First and foremost, future work should include a thorough and precise performance analysis especially on the improvement compared to the Python version. We believe that such analysis will require to carefully choose other application and to study their behavior. This analysis should also investigate the scalability consideration. To continue on the compression part, we believe that a design close to what has been done in Rezolus ([5], section Background) could improve the scalability of our design. The idea is to locally collect metrics at a high frequency and then accumulate the data over a moving window. The resulting data is then transmitted at a lower frequency to the collector. In our case, we could gather monitoring with this principle while keeping the current sending of all the data for the profiling.

In terms of permissions, we could have multiple group of metrics. Each group could be accessed by users with a certain permission. This could forbid users from modifying the metrics defined by the system administrators but also other users jobs. Building on the topic of security, we also note that it would be rather simple to add encryption of the data before sending on the network. But it would increase the overhead on the CPU of the compute nodes.

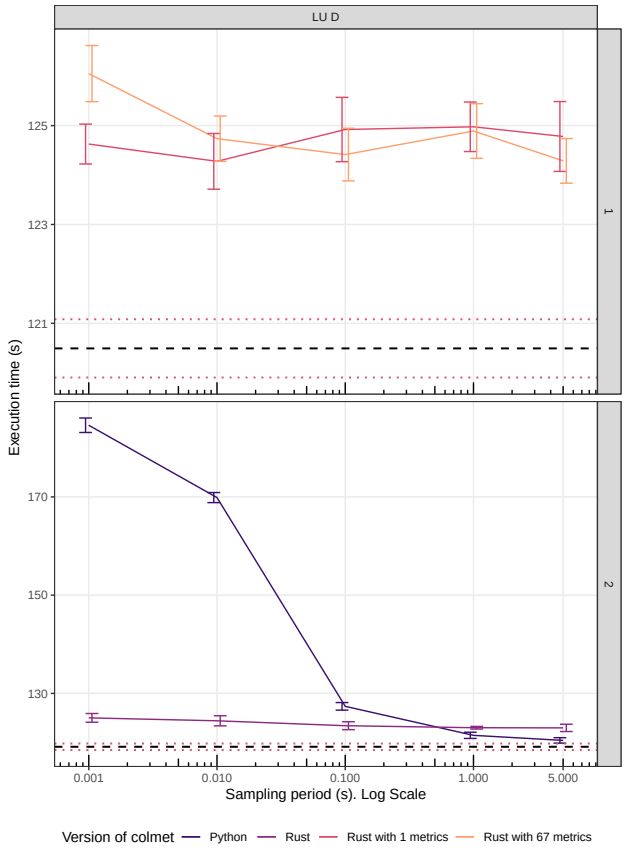
Another point that could be the object of future works is the integration with other monitoring or profiling tools. As detailed in Section 2, the subject of monitoring has been already extensively studied and many tools already implement many efficient way of collecting data. Future works could include finding a abstraction to include other tools as backends for Colmet. This would be a challenge in that other tools were not designed to support a different sampling period for each metric but it could allow Colmet users to benefit from works on other backends (not already implemented) or on data visualization and analysis (which could be useful for both monitoring and profiling).

6 Conclusion

In this paper, we presented the design and our implementation of Colmet, a tool capable of collecting metrics at different sampling period and to tag them with information on the job, allowing for job-scope monitoring and profiling with the same system. After giving design and implementation details, we presented the results of the performance analysis experiments performed. Experiments were carried out using multiple benchmarks from the NPB suite and studying the effect of the sampling period, of the number of metrics and of the version of Colmet on the execution time. The results show that the number of metrics does not cause a significant overhead on the execution of application. We also discussed some directions for future work.



(a) Overview of the 2 experiments (0 based)



(b) Zoom on the two LU experiments

Figure 2: Results of the experiments

The blacked dashed line is the reference without Colmet and the two red dashed line are the upper and lower value of the confidence interval.

References

[1] https://www.top500.org/lists/top500/2022/06/. Accessed the 09/06/2022.

[2] https://github.com/Meandres/colmet-rs. Accessed the 08/06/2022.

[3] https://github.com/oar-team/colmet. Accessed the 07/06/2022.

[4] https://github.com/Meandres/colmet-collector/blob/master/benchmarking/benchmark_colmet_rs.py. Accessed the 10/06/2022.

[5] https://github.com/twitter/rezolus/blob/master/docs/DESIGN.md. Accessed the 07/06/2022.

[6] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack Trace Analysis for Large Scale Debugging. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, Mar. 2007. ISSN: 1530-2075.

[7] A. Bartolini, F. Beneventi, A. Borghesi, D. Cesarini, A. Libri, L. Benini, and C. Cavazzoni. Paving the way toward energy-aware and automated datacentre. In *Proceedings of the 48th International Conference on Parallel Processing: Workshops*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[8] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In *The 6th IEEE/ACM International Workshop on Grid Computing, 2005.*, pages 8 pp.–, Nov. 2005. ISSN: 2152-1093.

[9] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194. IEEE, 2010.

[10] E. Dolstra, M. De Jonge, E. Visser, et al. Nix: A safe and policy-free system for software deployment. In *LISA*, volume 4, pages 79–92, 2004.

[11] J. Emeras, C. Ruiz, J.-M. Vincent, and O. Richard. Analysis of the jobs resource utilization on a production system. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–21. Springer, 2013.

[12] O. Flauzac, F. Mauhourat, and F. Nolot. A review of native container security for running applications. *Procedia Computer Science*, 175:157–164, Jan. 2020.

[13] E. Imamagic and D. Dobrenic. Grid infrastructure monitoring system based on Nagios. In *Proceedings of the 2007 workshop on Grid monitoring*, GMW '07, pages 23–28, New York, NY, USA, June 2007. Association for Computing Machinery.

[14] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[15] R. Mooney, K. Schmidt, and R. Studham. NWPerf: a system wide performance monitoring tool for large Linux clusters. In *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*, pages 379–389, Sept. 2004. ISSN: 1552-5244.

[16] A. Netti, M. Müller, A. Auweter, C. Guillen, M. Ott, D. Tafani, and M. Schulz. From facility to application sensor data: modular, continuous and holistic monitoring with dcdb. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–27, 2019.

[17] T. Röhl, J. Eitzinger, G. Hager, and G. Wellein. LIKWID Monitoring Stack: A flexible framework enabling job specific performance monitoring for the masses. *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 781–784, Sept. 2017. arXiv: 1708.01476.

[18] S. Saini and D. H. Bailey. NAS Parallel Benchmark Version 1.0 Results 11-96. page 53, 1996.

[19] S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *The International Journal of High Performance Computing Applications*, 20(2):287–311, May 2006.

[20] K. S. Stefanov, S. Pawar, A. Ranjan, S. Wandhekar, and V. V. Voevodin. A Review of Supercomputer Performance Monitoring Systems. *Supercomputing Frontiers and Innovations*, 8(3):62–81, Oct. 2021. Number: 3.

[21] B. Yang, X. Ji, X. Ma, X. Wang, T. Zhang, X. Zhu, N. El-Sayed, H. Lan, Y. Yang, J. Zhai, W. Liu, and W. Xue. End-to-end I/O monitoring on a leading supercomputer. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 379–394, Boston, MA, Feb. 2019. USENIX Association.