Bridging the gap between profiling and monitoring in HPC systems with dynamically reconfigurable fine-grain data collection

Ilya Meignan–Masson

Bridging the gap between profiling and monitoring in HPC systems with dynamically reconfigurable fine-grain data collection

Ilya Meignan--Masson
intern in the Datamove team, LIG, Univ. Grenoble Alpes

Supervised by: Olivier Richard

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

Abstract

Job-aware monitoring of HPC systems is a middle ground between system monitoring and profiling as it collects performance data at the scale of a job while keeping the overhead at a reasonable amount to allow for production usage. This paper presents Colmet, a job-level monitoring system capable of collecting metrics at different sampling period at the same time. The sampling periods can be dynamically reconfigured to suit the current need of the users up to the millisecond. We these features, Colmet can be used for usual monitoring purposes and for profiling purposes. Fine grain and reconfigurable data can also be used in systems that make decisions based on performance data. Our system is at least as efficient as state-of-the-art monitoring systems as we show in the performance analysis results.

1 Introduction

Modern HPC systems are composed of hundred of thousands if not millions of compute cores grouped in nodes. Such nodes are interconnected as well as connected to servers dedicated to storage with high performance networks. Cores are sometimes organized in complex Non-Uniform Memory Access (NUMA) architectures. Most modern systems also include accelerators such as GPUs, FPGAs, etc. They require large amounts of money, energy and effort for their design, their building and for their operation. At this scale, their efficient usage is an essential goal.

The scale and the complexity of those systems makes tedious any manual monitoring and motivates dedicated monitoring tools. Some systems focus on presenting to system administrators an overview of the status of components in the system (Nagios [12]). In this paper, we define monitoring as the process of collecting, gathering and storing data about the execution of one or several applications without modifying the source code or the executable of those applications. A monitoring system should allow users to use it online (*i.e.* while it is running) and this is the main difference with profilers that we define in a following paragraph. Some systems might include some way of displaying and presenting the collected data to the user. Others might be integrated in other systems that use the monitoring data to take decisions (*e.g.* in the Energy Aware Runtime [7]). We define a metric as a measurement of a property during the operation of a computer system. 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]. To answer the growing concern of energy-aware management of HPC systems, monitoring systems have included data collected from sources like the Running Average Power Limit (RAPL [8]) tool which exposes a collection of values related to energy consumption of the CPU. 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 store it in the most standard way while keeping an efficient and meaningful organization of the data.

In a cluster, users are allocated only a subset of the nodes in what is called a job. In the development process, they 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 3. 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 present Colmet, a dynamically reconfigurable tool capable of efficiently collecting metrics at a high frequency. With those features, our tool answers the profiling needs of users, the monitoring needs of system administrators and also developers of systems requiring very precise monitoring data.

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

2 Colmet : Design and Architecture

Colmet was first designed after the publication of [10] in 2013. After a first version in Perl, a version has been written in Python [2] that is available in production on Grid5000 clusters. To improve the performance, a Rust re-writing had been started but was left unfinished. The python version could not efficiently collect metrics at a small enough sampling period for profiling. 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 collection). At the moment only backends related to cgroups [11] and to `perf` events are implemented. We note here that the Python version is using the `taskstats` interface, which is a kernel interface to access data about a process and that our version uses metrics drawn from a `cgroups`. The scope of the monitoring is the same (*i.e.* job-level). The OAR RJMS is used in production on Grid5000 clusters. OAR creates a `cpuset` on the nodes allocated for the job isolating possible jobs running on the same machine. We leverage this behavior by collecting the data from the files created in the `cgroup` virtual filesystem. Finally, the Rust version also reuses Elasticsearch used as a time series database.

Colmet is composed of two components : a node agent and a collector. Figure 1 presents the architecture that we describe in the following paragraph.

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 for each job. 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. And other metrics can be set by the users for their job with possibly a higher sampling frequency, especially under the second. To this end, a script can be executed to update the list of metrics to collect or the default sampling frequency on a compute node. This update can only be done every second regardless of the collection sampling period to avoid the loss of performance when awaiting a potential update. Colmet automatically detects the creation or deletion of a job on a node by watching for the creation or deletion of a cgroup folder on the virtual file system. Default metrics can be defined which will be collected automatically on all the jobs except if configured otherwise by a user. The collection mechanism is as follow. Each time some metrics should be collected, the node agent queries the corresponding collection backends. The cgroup backend reads the data respectively in the files `cpu.stat` and `memory.stat` created in the virtual directory of each cpuset. For `perf` event data, they are also read from a file created by a call to `perf_event_open()`. Colmet keeps updated the list of the metrics with the time remaining until the next collection. 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, processes it and places it in the Elasticsearch database. To recover the name of the metric, the node agent and the collector share a list of metrics which allow for a coherent substitution. The fact that the collector is unique and that it requires a light yet significant processing means that the collector requires a dedicated server to run on. Communication between the node agents and the collector is done with ZeroMQ (ØMQ) sockets.

3 Related work

To begin with, we note the existence of a recent review of different monitoring systems for HPC systems [18]. This review identifies one of the problems of monitoring large scale HPC systems as requiring multiple tools with "custom scripts to get comprehensive monitoring of the HPC system" (section 3.1). We believe our work answers this issue by unifying in a single utility different aspects of the performance monitoring of an HPC system.

The monitoring of HPC systems is an extensively studied subject with a lot of tools available, with varying architectures and features. Two systems particularly relate to our work, namely Data Center Data Base (DCDB) [13] and the LIKWID Monitoring Stack (LMS) [15].

In terms of backends, the LMS, DCDB and Colmet implement the collection of metrics from various sources. The original and main backend of the LMS is hardware performance counters using the LIKWID toolkit. The LMS is also modular as it provides a library to allow other applications to send data to the storage. However, this library implies a modification of the code which is costly. DCDB, on the other hand, already integrates many collection backends ranging from operating system relate metrics to temperature and energy metrics as well as network and filesystem related metrics. Adding new backends is a possible task considering the relative modularity of DCDB but data cannot be pulled from

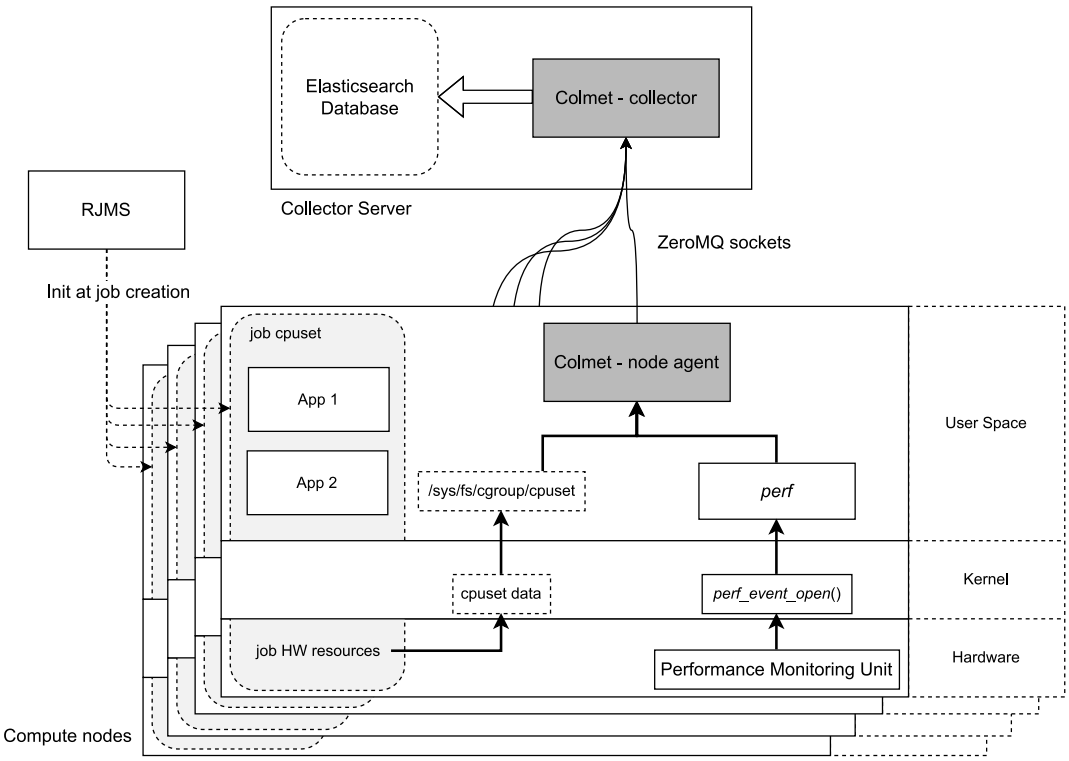


Figure 1: Architecture of Colmet

another tool. Colmet only implements hardware performance counters and operating system related backends and placed itself on the same level as DCDB in terms of modularity as adding backends requires adding a case in the Rust enumeration of backends.

Regarding the scope of monitoring, the LMS is close to Colmet in that it aggregate and store data at the job level. DCDB, on the other hand, stores data at the node level.

Colmet and DCDB supports having a different sampling period for each metric. The LMS, on the hand, partially supports it. The LIKWID toolkit is not capable of that. The only way of achieving such behavior would be to pull data from a collection of other tools each configured to a different sampling period. We believe that this would extremely tedious to set up as well as very inefficient.

Another feature is the implementation of ”front-end(s)” to help users to analyze and visualize the collected data. DCDB provides a set of analysis tools that can be run on each compute node before sending the data over the network or on the collector side before storing the data. The LMS on the other hand provide a Graphana dashboard to visualize the data. A similar dashboard have been developed for the Python version of Colmet but the compatibility with the Rust version have not been tested.

Other systems are worth mentioning as well. Beacon [19] and Examon [5] are focused on one particular backend, respectively IO/filesystem metrics and energy/temperature metrics. They also provide a deep analysis toolkit to allow users to improve their application while keeping their continuous monitoring use-case. On this sense, they are close to our motivation of mixing monitoring of the system and helping users in the performance analysis of their jobs.

In terms of profiling systems, Colmet is really far from toolkits like the Tuning and Analysis Utilities (TAU) [17] which can be used to instrument the code of distributed application for profiling. It relates more to systems like STAT [4] which attach to a MPI job and collect performance data of the execution.

To our knowledge, none of the existing systems implement the dynamic reconfiguration of the metrics to collect.

4 Performance analysis

We present in this section the methodology, the protocol and the results along with the analysis of the overhead of the monitoring and profiling system. We focus on the execution time overhead on the compute nodes. Our analysis is two fold. First we study our tool. Because users are given the possibility to reconfigure the tool, we measure the overhead of collecting more metrics. The second part is a more general comparison of Colmet with the previous python version as well as with other tools that we present in the related works section 3. As applications, we use 3 benchmarks from the NAS Parallel Benchmarks (NPB) suite. In this section, we describe our experimental set-up including the configuration of the different tools, present the results we obtained and the analysis that guided our exploration.

4.1 Experiment setup

Environment and applications

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. Finally, we tested the scaling with the Conjugate Gradient (CG) benchmark. This kernel uses a conjugate gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix (cf. the specification of the NPB [16], in Sections 3.1.1, 3.2.1 and 3.1.3). 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. We used class E for the EP benchmark and class D for the LU and CG benchmarks.

All our experiments were carried out in Grid5000 [6], more precisely in the dahu cluster of the Grenoble site. Each node 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). We used only the TCP network and not the Omni-Path that is available in this cluster. Nodes are running Debian 11 with OpenMPI 4.1.0. During the experiments, we used one MPI thread per core but not per hyper thread because it would not improve the performance as HPC application computations mainly involve floating point operations and there is only one Floating Point Unit (FPU) per core.

We used the Nix package manager during the experiments to install the required software [9]. This package manager provides strong guarantees on repeatability. 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.

We call configuration a set of values for the parameters. The experimental script requests a job, install the required softwares and executes a set of configuration specified in a YAML file. Because our second experiment compares multiple monitoring tools, an experiment requires multiple run of the script, one for each tool. To obtain results as accurate as possible, each configuration is replicated 30 times. We also randomize the order in which the configurations are executed to minimize the observational error. With the 30 values for each configuration, we compute the mean and the confidence intervals with a coefficient of 95%.

Configuration of monitoring tools

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 metrics 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 1 second up to the millisecond. Going beyond the millisecond would not be interesting for two reasons. First some of the monitoring tools do not support going below that value (*e.g.* Colmet Python). We also believe that lower values would cause a bottleneck on the network or simply that some system couldn't collect data at those higher frequencies because the time needed to collect data exceeds the sampling period.

Some monitoring systems require a extra server to run a collection agent. In our experiments, we simulate this behavior by including an extra compute node in the job reservation and making it play the role of the collector.

We compared our version of Colmet with 4 other configuration.

Colmet Rust This version is configured to collect metrics available on the virtual filesystem of the `cgroups` as well as hardware performance counters related metrics. Some experimental configurations corresponds to collecting all the metrics possible which in turn means that exactly 28 hardware performance counters related metrics are collected. This is too much to fit in the registers dedicated to performance counters in any CPU on the market resulting in a significant overhead [14]. Doing so is meaningful in our analysis as this is the worst case. Another point worth noting is that we do not use the reconfiguration tool to change the number of metrics nor to change the sampling period. Like the rest of the tools, the script kills the monitoring system and spawns another instance with other parameters.

Without The trivial one is the reference without any monitoring system running.

Colmet Python The first actual system is the previous version of Colmet written in Python. This tool is configured to collect metrics exposed by Linux under the virtual filesystem `/proc`, `taskstats` metrics as well as the metrics available on the virtual filesystem of the `cgroups`.

LIKWID This value is measured using the `likwid-perfctr` executable running on each of the nodes. It is configured to collect the performance group `FLOPS_DP` and in the timeline mode which collects data at fixed intervals. While LIKWID provides a wrapper around `mpirun` that launches `likwid-perfctr` instances before passing the control to MPI, this executables doesn't allow for periodic sampling and thus is not relevant to our analysis.

DCDB This tool requires a set of configuration files. The directory of the script includes a base configuration and the actual configurations are generated with the values in the YAML configuration file using this base configuration. The base configuration includes metrics from the `/proc` virtual filesystem as well as from the `/sys` virtual filesystem. The script launches the executable `dcdbpusher` on all the nodes with the relevant configuration and `collectagent` on the extra node.

As DCDB uses MQTT as data broker and Cassandra database as a storage backend, an instance of Mosquitto which is an implementation of MQTT and of Cassandra is launched on the extra node.

4.2 Overhead of the number of metrics

The goal of this experiment is to compare the overhead of collecting more metrics. The two configurations are running Colmet (Rust version) with 3 metrics, one for each of the backends, and with 67, *i.e.* every possible metrics. The results of this experiment are presented in Fig.2a, Fig.2b and Fig.2c.

The results show that for the EP benchmark, there is no significant difference between the two configurations. For the LU and the CG benchmark, there is no significant difference except at a sampling period of 1 ms.

For the LU benchmark at 1 ms of sampling period, the overhead is of 9,6 $\pm$ 0.5 %. For the CG benchmark at 1 ms, the overhead is of 9.6 $\pm$ 0.06 % for 4 nodes, 8 $\pm$ 0.4 % for 8 nodes and 7 $\pm$ 0.04 % for 16 nodes.

4.3 Comparison of the overhead of various monitoring systems

The goal of this experiment is to compare the overhead of our system, of the previous version of Colmet as well as of some state of the art monitoring systems. Note that the Rust version of Colmet is configured to collect all the 67 metrics possible meaning as we stated in the previous section that this is the upper bound. The results of this experiment are presented in Fig.3a, Fig.3b and Fig.3c. Values are presented in Tables 1, 2 and 3.

These plots are normalized with 1 being the reference without any monitoring system running. Considering each configuration as a random variable, the variable of any configuration with a monitoring tool is dependent with the variable of the reference configuration. For obvious reasons, the variance of execution time of the benchmark will impact the execution time of the combination (benchmark + monitoring). However, for the benchmarks LU and CG, the coefficient of variation is negligible (A) meaning that computing the mean of the division of two random variable is equal enough to the division of the means of the two variables. For the EP benchmark, on the contrary, the coefficient of variation is approximately 15%. For this benchmark, Fig.3a should be taken as indicative and values should not be used as is. Raw results presented in A are used in the following paragraph and should be used for any analysis. As stated previously, the variance of the results with the EP benchmark is quite large. We believe that this is due to the EP being compute-intensive.

Based on the experimental results, we see that our tool is significantly equivalent to state-of-the art tools at normal sampling period across all the benchmark tested. In smaller sampling period, it performs even better than the other tools. As an example, at a sampling period of 1 ms, the execution time is 207 $\pm$ 1.98 s where the best other tool is DCDB configured with 32 threads at 216 $\pm$ 9.99 s for the EP benchmark. For the LU benchmark, both LIKWID and Colmet Python are close to 65% of overhead while Colmet Rust is at 17.1 $\pm$ 0.6% of overhead. Finally, at 1 ms of sampling period, the CG benchmark results shows that both DCDB (32 threads) and our tool have a significantly equivalent overhead which is lower than any of the other tools. The same tools have a significantly negligible overhead across any of the other configurations.

To conclude our analysis, the data collected using the protocol described before hand show that our tool has a lower overhead than most of the other state of the art tools from usual sampling period up to the millisecond. Except at the millisecond period, the overhead of our tool is also significantly negligible compared to the reference only with the benchmark running. The results of the first experiment also show that the number of metrics has a negligible impact on the execution time of our tool except at the millisecond scale.

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 agent completely.

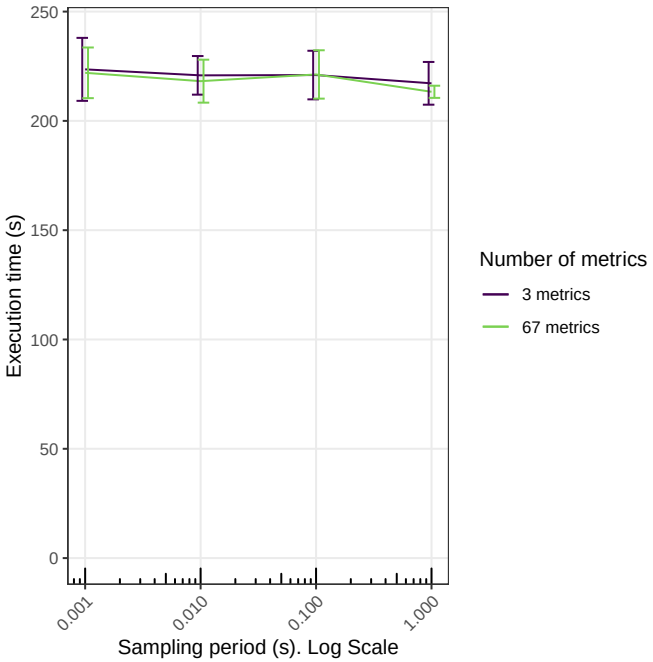
Another limitation on the design is the scalability. All our experiments were using a relatively small number of nodes with regard to the usual size of small to middle size clusters. At a larger scale, our choice of simple compression might not be enough to prevent an important overhead on the network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

5.2 Directions

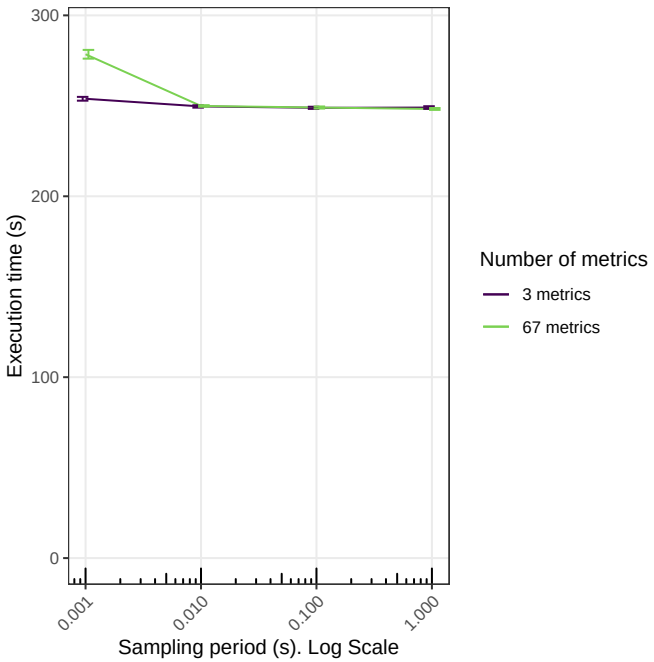
To continue on the compression part, we believe that a design close to what has been done in Rezolus [3] (section Background) could improve the scalability of our design. The idea is to locally collect metrics at a high frequency and then accumulate the data over a moving window. The resulting data is then transmitted at a lower frequency to the collector. In our case, we could implement a switch between sending raw data or accumulating locally on the nodes to allow for flexibility depending on the use-case.

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.

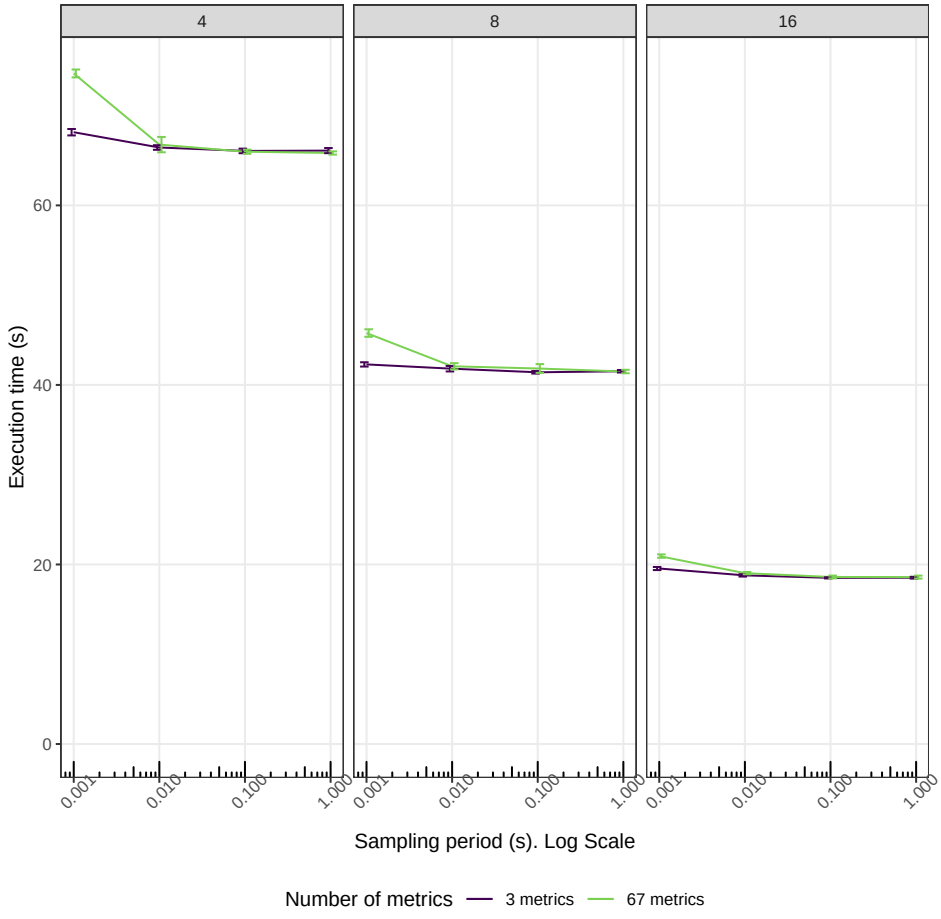
Another point that could be the object of future works is the integration with other monitoring or profiling tools. As detailed in Section 3, the subject of monitoring has been already extensively studied and many tools already implement



(a) Performance comparison for the EP benchmark. The experiment is performed on 4 nodes.

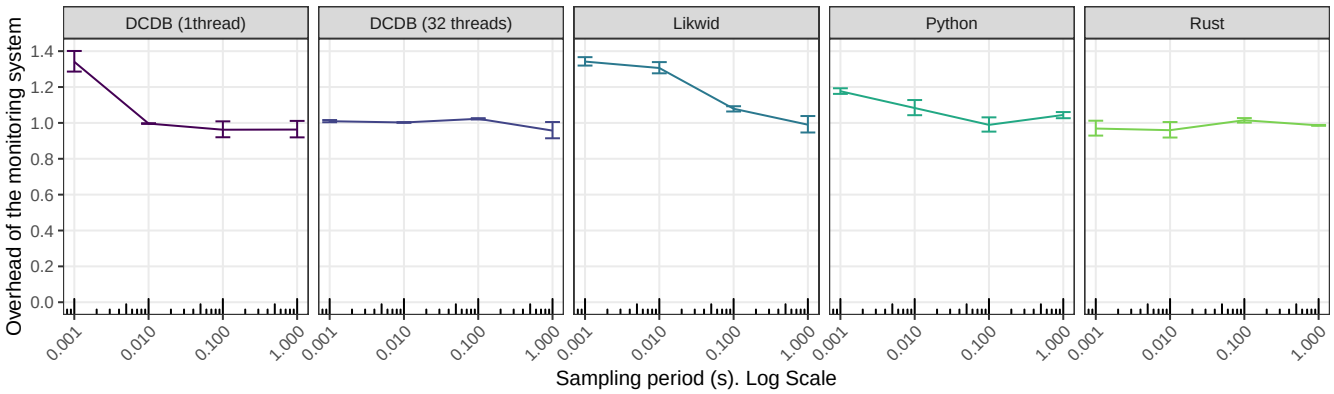


(b) Performance comparison for the LU benchmark. The experiment is performed on 4 nodes.

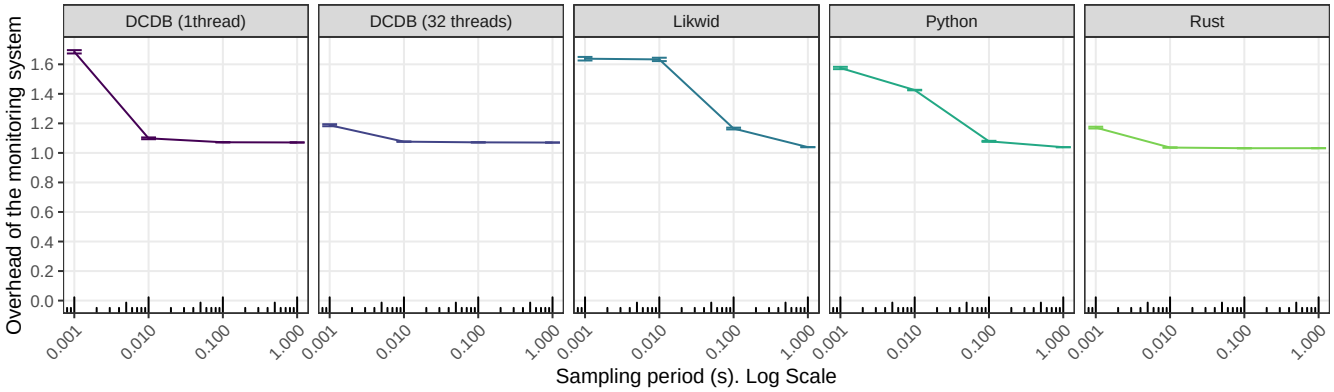


(c) Performance comparison for the CG benchmark. The experiment is performed on 4, 8 and 16 nodes.

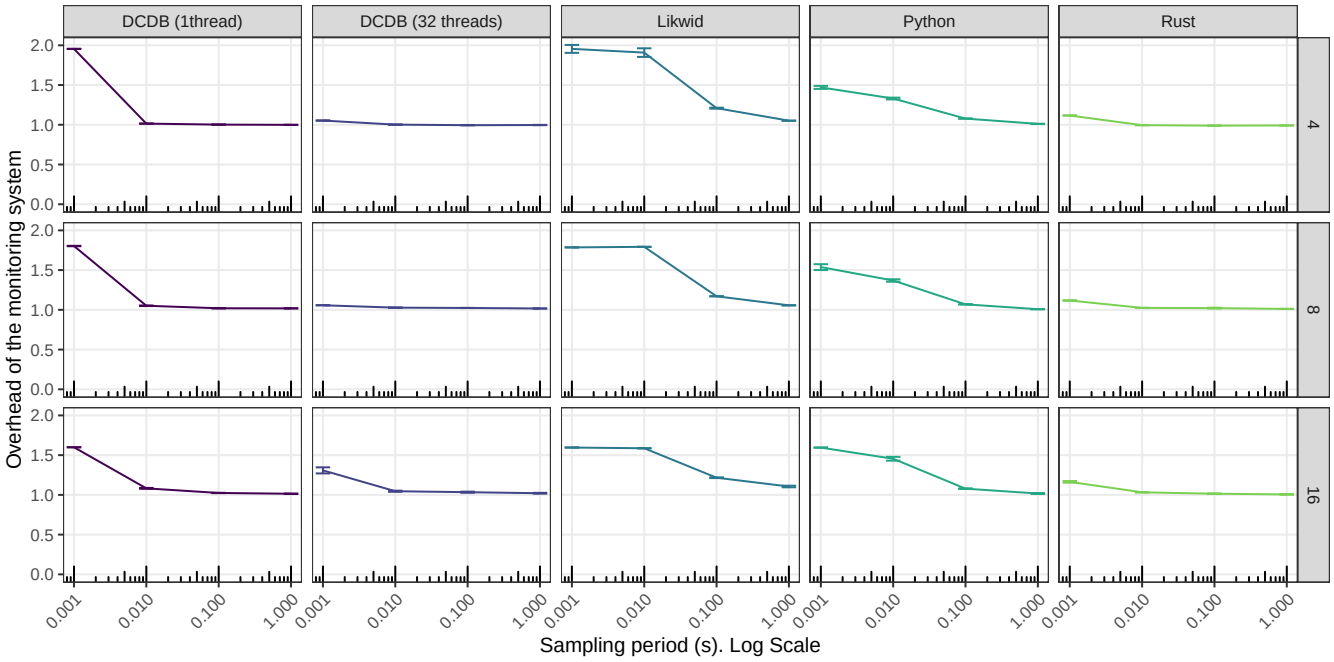
Figure 2: Comparison of Colmet (Rust) configured to collect 3 metrics and Colmet (Rust) configured to collect 67 metrics. The confidence intervals are computed at 95%.



(a) Performance comparison for the EP benchmark. The experiment is performed on 4 nodes.



(b) Performance comparison for the LU benchmark. The experiment is performed on 4 nodes.



(c) Performance comparison for the CG benchmark. The experiment is performed on 4, 8 and 16 nodes.

Figure 3: Comparison of different monitoring systems presented in Section 3. 1 represents the time only with the benchmark. The configuration of these systems is presented in 4.1 As described in 4.3, the ratio for the EP benchmark should be taken as indicative. See A for raw numbers. The confidence intervals are computed at 95%.

many efficient way of collecting data. Future works could include finding a abstraction to include other tools as backends for Colmet like it is done in [15]. This would be a challenge in that other tools were not designed to support a dynamically changed sampling period 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 for each metric and dynamically reconfigurable sampling period. The system tags them with information on the job, allowing for job-scope monitoring and profiling with the same system. After describing the architecture of the system as well as the backends used to collect data, we presented our performance analysis experiments. The impact of the number of metrics on the execution time and the overhead of different monitoring systems compared to the reference are presented. The results show that the number of metrics only has an significant impact at the millisecond scale and only for 2 of the 3 benchmarks tested. Our system has a significant overhead compared to running without only at the millisecond scale. It outperforms the other systems we compared it to only being equal to DCDB configured with 32 collecting threads. Finally, we discussed limitations and future work.

References

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

[2] https://github.com/oar-team/colmet. Accessed the 23/08/2022.

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

[4] 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.

[5] 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.

[6] 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.

[7] J. Corbalan and L. Brochard. Ear: Energy management framework for supercomputers. In *Barcelona Supercomputing Center (BSC) Working paper*. 2019.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] A. Nowak and G. Bitzes. The overhead of profiling using pmu hardware counters. Jul 2014.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

A Appendix : Experimental results

Table 1: Results for the EP benchmark with 4 nodes

monitoring software	sampling period	mean execution time	error
Without	-1	214.58	11.2
DCDB (1thread)	0.001	287.67	2.72
DCDB (32 threads)	0.001	216.61	9.99
Likwid	0.001	287.95	10.01
Python	0.001	252.48	9.84
Rust	0.001	207.83	1.98
DCDB (1thread)	0.01	213.81	11.53
DCDB (32 threads)	0.01	215.07	10.84
Likwid	0.01	280.28	7.93
Python	0.01	232.41	3.09
Rust	0.01	205.87	1.5
DCDB (1thread)	0.1	206.43	1.29
DCDB (32 threads)	0.1	219.49	12.24
Likwid	0.1	231.55	15.22
Python	0.1	212.24	2.58
Rust	0.1	217.79	14.06
DCDB (1thread)	1	206.61	0.97
DCDB (32 threads)	1	205.39	1.02
Likwid	1	212.44	1.27
Python	1	224.08	15.36
Rust	1	211.61	10.55

Table 2: Results for the LU benchmark with 4 nodes

monitoring software	sampling period	mean execution time	error
Without	-1	231.8	0.39
DCDB (1thread)	0.001	390.5	3.27
DCDB (32 threads)	0.001	275.26	2.06
Likwid	0.001	379.64	3.45
Python	0.001	365.08	2.44
Rust	0.001	271.51	1.87
DCDB (1thread)	0.01	254.7	1.84
DCDB (32 threads)	0.01	249.45	0.56
Likwid	0.01	378.54	3.28
Python	0.01	330.42	0.97
Rust	0.01	240.2	0.42
DCDB (1thread)	0.1	248.52	0.36
DCDB (32 threads)	0.1	248.36	0.57
Likwid	0.1	270.06	1.86
Python	0.1	249.97	1.22
Rust	0.1	239.18	0.35
DCDB (1thread)	1	248.19	0.49
DCDB (32 threads)	1	248.06	0.44
Likwid	1	240.86	0.45
Python	1	240.78	0.54
Rust	1	239.22	0.5

Table 3: Results for the CG benchmark with 4, 8 and 16 nodes

monitoring software	nb nodes	sampling period	mean execution time	error
Without	4	-1	66.11	0.3
DCDB (1thread)	4	0.001	129.21	0.47
DCDB (32 threads)	4	0.001	69.59	0.5
Likwid	4	0.001	129.22	3.88
Python	4	0.001	97.19	1.67
Rust	4	0.001	73.8	0.35
DCDB (1thread)	4	0.01	67.06	0.3
DCDB (32 threads)	4	0.01	66.21	0.17
Likwid	4	0.01	126.13	4.16
Python	4	0.01	87.94	1.14
Rust	4	0.01	65.74	0.27
DCDB (1thread)	4	0.1	66.24	0.23
DCDB (32 threads)	4	0.1	65.74	0.22
Likwid	4	0.1	79.91	0.78
Python	4	0.1	71.24	0.32
Rust	4	0.1	65.37	0.21
DCDB (1thread)	4	1	66.03	0.23
DCDB (32 threads)	4	1	65.88	0.18
Likwid	4	1	69.5	0.28
Python	4	1	66.85	0.3
Rust	4	1	65.53	0.28
Without	8	-1	40.97	0.23
DCDB (1thread)	8	0.001	73.86	0.32
DCDB (32 threads)	8	0.001	43.33	0.24
Likwid	8	0.001	73.16	0.36
Python	8	0.001	63	1.87
Rust	8	0.001	45.78	0.41
DCDB (1thread)	8	0.01	43.07	0.39
DCDB (32 threads)	8	0.01	42.09	0.14
Likwid	8	0.01	73.45	0.35
Python	8	0.01	56.09	0.95
Rust	8	0.01	41.97	0.22
DCDB (1thread)	8	0.1	41.78	0.14
DCDB (32 threads)	8	0.1	41.93	0.2
Likwid	8	0.1	47.96	0.29
Python	8	0.1	43.79	0.13
Rust	8	0.1	41.86	0.46
DCDB (1thread)	8	1	41.73	0.18
DCDB (32 threads)	8	1	41.66	0.23
Likwid	8	1	43.29	0.14
Python	8	1	41.3	0.17
Rust	8	1	41.45	0.29
Without	16	-1	18.53	0.07
DCDB (1thread)	16	0.001	29.64	0.1
DCDB (32 threads)	16	0.001	24.25	0.79
Likwid	16	0.001	29.58	0.13
Python	16	0.001	29.57	0.14
Rust	16	0.001	21.57	0.27
DCDB (1thread)	16	0.01	20.06	0.18
DCDB (32 threads)	16	0.01	19.39	0.2
Likwid	16	0.01	29.41	0.08
Python	16	0.01	26.96	0.54
Rust	16	0.01	19.14	0.13
DCDB (1thread)	16	0.1	18.99	0.1
DCDB (32 threads)	16	0.1	19.17	0.2
Likwid	16	0.1	22.54	0.16
Python	16	0.1	19.99	0.14
Rust	16	0.1	18.81	0.12
DCDB (1thread)	16	1	18.8	0.12
DCDB (32 threads)	16	1	18.92	0.16
Likwid	16	1	20.49	0.26
Python	16	1	18.86	0.18
Rust	16	1	18.64	0.13