uc3m | Universidad Carlos III de Madrid

e-Archivo

This is a postprint version of the following published document:

A. Cascajo, D. E. Singh and J. Carretero, "LIMITLESS - LIght-weight MonItoring Tool for LargE Scale Systems," *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2021, pp. 220-227.

DOI: 10.1109/PDP52278.2021.00042

©2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

LIMITLESS - LIght-weight MonItoring Tool for LargE Scale Systems

Alberto Cascajo
University Carlos III of Madrid
Leganés, Madrid, Spain
acascajo@inf.uc3m.es

David E. Singh
University Carlos III of Madrid
Leganés, Madrid, Spain
dexposit@inf.uc3m.es

Jesus Carretero
University Carlos III of Madrid
Leganés, Madrid, Spain
jcarrete@inf.uc3m.es

***Abstract*—This work presents LIMITLESS, a HPC framework that provides new strategies for monitoring clusters. LIMITLESS is a scalable light-weight monitor that is integrated with other HPC runtimes in order to obtain an holistic view of the system that combines both platform and application monitoring. This paper presents a description of the novel components of the architecture, including new approaches for reaching a higher scalability based on a combination of in-transit processing and performance prediction. This work also includes a practical evaluation on simulated and real platforms, that shows significant monitoring scalability, retrieving data capacity and reduced overheads.**

***Index Terms*—Monitoring, application modelling, performance prediction.**

I. Introduction

Currently, one of the key challenges in large-scale clusters is to determine as accurately as possible the status of the system every small fraction of time. There are two main approaches for obtaining this information: by means of the compute-node monitoring or by means of the analysis of the applications that running on the platform. The first one involves a system-wide monitoring infrastructure while the second one is usually restricted to the use of monitoring software executed with the applications. In this work we combine these two approaches in order to provide, not only a more accurate cluster monitoring, but also a scheme that permits to model the application behavior in order to reduce the infrastructure monitoring overhead.

This work presents LIMITLESS, a highly-scalable monitor that is able to work under near-to-second sampling rates. LIMITLESS is fully integrated with other system software like the scheduler, CLARISSE and FlexMPI runtimes to enhance the monitor scalability. CLARISSE [1] is a middleware for data-staging coordination and control on large-scale HPC platforms that provides application I/O monitoring. FlexMPI [2] provides malleable capabilities to MPI applications and performs application CPU monitoring. In this way, the main contributions of this work are:

- An integration between LIMITLESS monitor and other runtimes for collecting application-related information.

This work was partially supported by the European Union's Horizon 2020 ASPIDE project (grant agreement No 801091), and the Spanish Ministry of Science and innovation Project DECIDE (Ref. PID2019-107858GB-I00.)

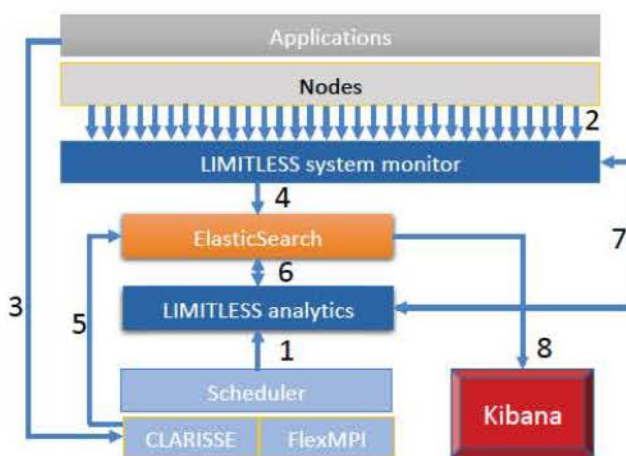

Fig. 1. General overview of the system architecture and interrelation with other components.

- A novel analytic module for application performance modeling.
- A novel in-transit processing scalable feature for reducing the monitor traffic.
- A practical evaluation of the complete infrastructure on both simulated and real platforms.

The structure of the paper is as follows: Section II describes the architecture organization; Section III describes LIMITLESS's features for providing monitoring scalability; Section IV presents the different algorithms used to predict the immediate future state of the cluster; Section V provides a practical evaluation of the LIMITLESS monitor and the analytic tools provided; Section VI shows relevant works related to our proposal. Finally, Section VII summarizes the main conclusions and future work.

II. Monitor Architecture

LIMITLESS is a light-weight scalable monitor that operates on each compute node and provides information about the platform resources and applications that are being executed. Figure 1 shows a general overview of how LIMITLESS is integrated with other system components like the application scheduler, FlexMPI and CLARISSE runtimes. As shown in this figure, LIMITLESS includes four main components: a *System monitor* that collects the performance metrics from the

cluster, an *ElasticSearch* database [3] that provides persistent storage, Kibana, a *GUI* for displaying the cluster information in an user-friendly format, and an *Analytic* component that is used to analyse and model the executing applications.

Limitless Analytics (LAN) is the component that deals with the storage, visualization, communication with the scheduler and is the responsible of the performance prediction. It stores and manage the application models, generates the predictors, trains and executes the neural networks and the machine learning algorithms, and, in addition to that, the LAN component includes an API based on sockets for plugging other programs or modules, including user programs or other data visualizers.

When one application is executed, the scheduler notifies LIMITLESS Analytics (arrow 1) about the application characteristics (which is used to identify and classify the application). In a second step, when the applications are executed two different metrics are collected simultaneously: at node level to the monitor (arrow 2) and at application-level to FlexMPI and CLARISSE (arrow 3). Then, both metrics are processed by the respective runtimes and are written into Elastic search (arrows 4 and 5). Note that this process is concurrent. Then, the LIMITLESS analytics creates an application model using the information stored in ElasticSearch (arrow 6). And finally, the prediction model (arrow 7) is sent to LIMITLESS in order to perform in-transit processing for reducing the network monitoring traffic. During all these processes, Kibana may be used to visualize (arrow 8) the cluster status.

A. System monitor

LIMITLESS is a monitoring tool designed to provide performance information for generic purposes in large scale systems. One of its main interesting features is the possibility of change the monitoring period (also called *sample interval*) online, having one different for each node, and without the necessity of restart the system or the monitor. The monitoring interval can be set in a range of time from hours to seconds and also sub-second.

The System Monitor consists of one *LIMITLESS Daemon Monitor* (LDM) per node, which periodically collects the performance metrics; a set of *LIMITLESS DaeMon Aggregators* (LDAs), that forwards the information from the LDMs to other aggregators or servers; and the *LIMITLESS DaeMon Server* (LDS) that gathers and stores the monitoring information in ElasticSearch. Figure 2 shows a typical deployment of LIMITLESS with replication of the LDAs and LDSs. The purpose of replicating these components is to enhance the monitor scalability and resilience. The monitoring information collected by LIMITLESS includes different per-compute-node metrics related to CPU, GPU, cache memory, I/O, and network usage as well as energy consumption.

B. Monitoring policies

The monitoring policies specify how the performance metrics are collected by the LDS. There are two different alternatives: continuous monitoring and event-based monitoring. In the first one each LDM sends all metrics collected (usually in one network packet) every sample interval for obtaining an updated cluster status. The second one is not designed to provide a global vision of the cluster, but to notify events in certain compute nodes that are relevant or could harm the platform performance. We assume that an event occurs when a given performance metric (such as CPU energy usage) is above a certain user-defined threshold. Based on this, LDM will transmit only monitoring metrics that are bigger than the threshold. Note that this policy reduces the network traffic, as the system will only receive a notification when an event is produced. These notifications are displayed in the GUI and also sent to the scheduler, allowing to enhance the application scheduling [4].

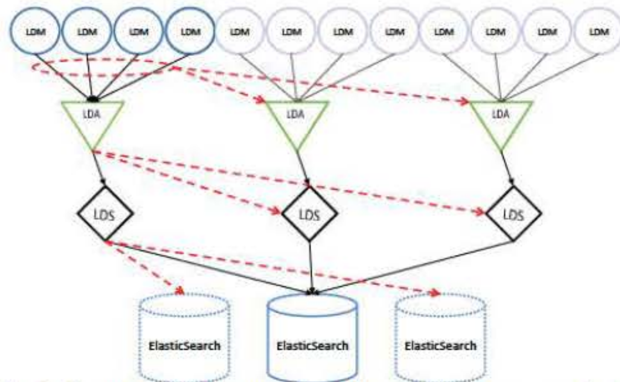

Fig. 2. Example of deployment with TMR and watchdog processes in the first monitoring branch.

C. Framework Deployment

Generic topologies can be used to deploy LIMITLESS. This permits to adapt the monitor deployment with the same topology as the cluster's communication network. Figure 2 shows an example of a particular deployment with four compute-nodes per LDA, one LDA per LDS, and one ElasticSearch database. In addition to the topological deployment, each component (for instance, a LDM) can be dynamically configured to adapt to the application characteristics (for instance, reducing the sampling interval in some nodes with a more variable workload) or to dynamically reconfigure the monitor connection topology.

D. Fault tolerance

LIMITLESS includes two different policies that are complementary: triple modular redundancy and watchdog. The first one allows the monitor components to be connected with a set of three different next-level components: each LDM is connected with three LDAs; each LDA is connected with another three LDAs and/or LDSs; and each LDS is connected with three ElasticSearch databases.

The watchdog policy provides resilience by launching each component with a service that checks whether the node works properly. In case of a component failure, the monitor will attempt to rerun the component with the same configuration. If this is not possible, the node is flagged as non-operational. Figure 2 shows the fault tolerance mechanisms in red colour.

The dotted red lines shows the triple module redundancy configuration for each node.

E. Data Storage and Visualization

ElasticSearch is used to store the data produced by the framework. Main features of ElasticSearch are high performance processing, its suitability for big data storage, compression support and prebuilt data replications. Note that it is configured to deal efficiently with large network packets, so with this approach we aim to leverage this feature efficiently. Moreover, ElasticSearch is integrated with Kibana [5], an open-source scalable web interface for visual data representation. Kibana provides plugins to manage the data stored in the database, creating dashboards, performing time-series analysis and applying machine learning algorithms to the stored data.

F. CLARISSE and FlexMPI support

The execution framework shown in Figure 1 includes CLARISSE and FlexMPI runtimes. CLARISSE main goal is to provide control mechanisms through the I/O software stack in order to enhance the application I/O by means of coordinated policies. FlexMPI brings malleable capabilities to MPI applications that are able to dynamically increase or reduce the number of processes. Both runtimes include libraries that are linked and executed with the application. These libraries that monitor the application I/O activity (by means of CLARISSE) and the application CPU and memory usage (by means of FlexMPI). The performance metrics are sent to the corresponding external controllers for CLARISSE and FlexMPI and subsequently are stored in ElasticSearch. By means of this approach it is possible to include fine-grained information about the application characteristics. For instance, it is possible to record the duration of each I/O phase with a precision of milliseconds. This information will be used for carrying out a more precise application modelling that will be subsequently used by the monitor.

III. Scalable support

This section describes the three functionalities included in LIMITLESS for enhancing the monitoring scalability: optimized Data Packing, in-situ and in-transit processing. Optimized Data Packing is performed by the LDMs. In order to maximize the efficiency of this process, architecture-dependent features like the maximum packet length, are taken into account to optimally group different metrics into a single network packet by means of bit-level codifications that results in a significant reduction in the packet size.

LDMs include in-situ processing algorithms [4] that analyse the captured metrics and selectively avoid transmitting them when the difference regarding the previous metrics is under a predefined threshold. By means of this approach, the amount of monitoring traffic related to a steady compute node can be reduced while guaranteeing a continuous monitoring of the system. Note that this algorithm is light-weight, given that it is executed in the same compute nodes as the applications.

Algorithm 1 LIMITLESS in-transit pseudocode executed by the Analytic components. A_i stands for the application context of application i-th. E is the ensemble related to the application. P_i stands for the application's performance predictor.

```
1: // Limitless Analytic
2: INPUT (from scheduler): A_i
3: E = get_ensemble(A_i)
4: if E == ∅ then
5:     E = create_ensemble(A_i)
6: else
7:     P_i = generate_predictor(m_i, E)
8:     send_predictor(P_i)
9: end if
```

Algorithm 2 LIMITLESS in-transit pseudocode executed by the LDA. P_i stands for the i-th application's performance predictor.

```
1: // Limitless DaeMon Aggregator (LDA)
2: INPUT (from analytics): P_j, j = 1,... N_app
3: while TRUE do
4:     for (i = 0; i < N_app; i++) do
5:         m_i = get_metrics(A_i)
6:         if P_i == ∅ then
7:             send_metrics(m_i)
8:         else
9:             n_i = generate_metrics(P_i)
10:            if ||n_i − m_i|| < threshold then
11:                do_nothing
12:            else
13:                send_metrics(m_i)
14:            end if
15:        end if
16:    end for
17: end while
```

In [4] we have shown that by means of this approach it is possible to reduce up to the 87% of the metrics sent from the LDMs to the LDAs.

In-transit processing is based on using prediction algorithms in the LDA components. Figure 3 shows an example of this strategy which leverages the integration between LIMITLESS and scheduler for developing a prediction algorithm based on the application characteristics. For each new executing application, the scheduler notifies the monitor (arrow 1) the application name, input arguments and compute nodes that have been allocated. We denote it as *application context.* Based on that, a unique application ID is created and used for indexing all the performance metrics related to this execution. These metrics, along with the identifier and application context, are stored in Elastic Search.

In many cases, during the application development, the applications are executed several times with minor changes in its configuration. We call this collection of executions *application ensemble*. Note that the executions belonging to the same ensemble have similar characteristics, like the duration of the CPU and communication phases, or the I/O access pattern. LIMITLESS includes an *Analytic component* that is able to identify the application ensemble related to

Algorithm 3 LIMITLESS in-transit pseudocode executed by the LDS. P_i stands for the i-th application's performance predictor.

```
1: // Limitless DaeMon Server (LDS)
2: INPUT (from analytics): P_j, j = 1,... N_app
3: while TRUE do
4:    for (i = 0; i < N_app; i++) do
5:       if new_metrics(A_i) then
6:          m_i = get_metrics(A_i)
7:          write_metrics(m_i)
8:       else
9:          n_i = generate_metrics(P_i)
10:         write_metrics(n_i)
11:      end if
12:   end for
13: end while
```

each new executing application. This is done by comparing the application execution context with the existing ones stored in Elastic Search. Algorithm 1 shows the Analytic component pseudocode. For every newly executed application A_i the related ensemble is obtained (line 3). If it doesn't already exist, a new one is created (line 5). Otherwise, once the ensemble is identified, the Analytic component compares (line 7) the collected application metrics with the ones related to previous executions (that are stored in Elastic Search). If they are similar (in terms of duration of the CPU, communication and I/O phases) a predictor is generated taking into account both the current and previous executions. This predictor is sent (line 8) to the LDAs and LDSs involved in the application execution[1].

Figure 3 illustrates an overview of this process for an application (denoted as App) that is executed several times. We denote A_i with $i = 1, \ldots n$ each one of the related execution context i. When the application is executed for the first time, a new application ensemble is created, $E = \{A_1\}$. Then, before starting the second execution, the related application context sent by the scheduler (arrow 1) permits the Analytic component to identify if it belongs to any existing ensemble E based on the Elastic Search records (arrow 2), which is properly updated $E = \{A_1, A_2\}$ and the generated model is created and sent to the corresponding LDAs and LDS (arrow 3). When the application is executed, the LDAs compare the incoming metrics (arrow 4) from the LDM with the ones predicted by the model (arrow 5). If the metrics are the same (within a given threshold), these metrics are not sent (arrow 6) because the LDS is able to generate them (arrow 7) using the previous model.

Algorithm 2 shows the logic related to the LDAs. When the A_2 execution starts, the performance metrics are collected (line 5). If the performance predictor does not exist, then the metrics are sent to the LDS (line 7). Otherwise, the metrics are compared with the generated ones by the model (line 10). If they are the same (within some limits), they are not

[1]Note that the specific list of LDAs and LDSs related to the application is obtained from the scheduler which determines the specific compute nodes where the application is executed.

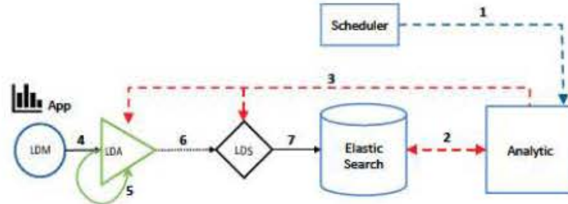

Fig. 3. LIMITLESS - Network traffic reduction thanks to modeling and prediction.

Algorithm 4 Application performance model logic based on pattern matching. Variables n_i and m_i corresponds, respectively, to the ith measured and recorded performance metrics.

```
1: // Limitless Analytic
2: INPUT (from Elastic search): m_i
3: {n_i} = ES_read_metrics(i)
4: if ||n_i - m_i|| < threshold then
5:    n_{i+1} = ES_read_metrics(i + 1)
6:    return(n_{i+1})
7: else
8:    return("Predictionfailed")
9: end if
```

sent to the LDS (line 11) saving network traffic. In the other case, the predictor fails to reproduce the application behaviour. Consequently, the prediction is overridden and the metrics are sent to the LDS (line 13) until a more accurate model is generated.

Algorithm 3 shows the logic related to the LDSs. The basic idea behind this algorithm is that if the prediction fails in the LDAs, performance metrics should arrive from these components. In this way, the LDSs waits for incoming metrics (line 5). If they arrive, they are taken and stored in Elastic Search (lines 6 and 7). Otherwise, the metrics are generated using the model and stored (lines 9 and 10). Note that the performance prediction P_i used by the LDAs and LDSs is the same.

IV. PERFORMANCE PREDICTION TECHNIQUES

This section describes the different algorithms used to predict the immediate future state of the cluster. The idea of this component is to leverage the large amount of data that LIMITLESS collects for predicting the future behaviour of the applications. We have used three different prediction techniques based on single-variable analysis (application pattern matching, prediction based on historical window, and classification based on neural networks) and one based on multi-variable correlation (based on machine learning).

The first technique (depicted in Algorithm 4) is based on pattern generation. This solution stores the performance metrics as a pattern during an execution of an application. In next executions, each collected metric is compared with the corresponding metric in the pattern. If the difference is below a given threshold, then the following metric is generated from the next recorded one. Otherwise, the model fails to provide

a prediction. This model is lightweight and works well with applications that exhibit the same execution pattern in different executions.

The second technique corresponds to a short-time predictor based on a sample-window. This algorithm predicts the performance values based on the interpolation of a certain amount of previous values. The size of the set used in the interpolation is proportional to the size of the window. For instance, a window of size 5 uses the last recorded five samples to compute the interpolation and produce each prediction. Note that this approach works well for periodic applications with different phases (CPU, communication, I/O), where the length of the phases is similar during the execution and there are little variations in the performance metrics during each phase. The main different aspect of the first solution is the utilization of an interpolation algorithm that is used to refine the predicted values, instead of using directly the next pattern value.

The third technique uses neural networks to learn the application behaviour from data coming from many previous executions and to predict the immediate states when a known application is deployed. We have used one neural network per application which is trained with the monitor data of the application ensemble. Each neural network is composed of three layers with 120, 60 and 1 neurons respectively. The training starts when the first application of a given ensemble is performed. After that, each network is capable of adjusting its weights to recognize the application pattern. The second application execution is used to assess the accuracy of the prediction. If it is not accurate enough, further executions are used to provide more training data. Note that the main advantage of this approach is that it is not necessary to use historical values for making a prediction.

The last prediction technique is based on a multi-variable correlation using the Nearest Neighbour Machine Learning algorithm [6] [7]. In this approach, given a set of k performance metrics collected by a LDM in a current sample, this algorithm finds the most similar k-metrics to this set in the complete historical data. Then, the following recorded ones are provided as prediction. Note that this algorithm is able to make predictions using very large datasets (like the historical records related to an application ensemble) with a low overhead. The idea behind this approach is to leverage the similarity between different metrics belonging to the same application ensemble to make a prediction.

V. Evaluation

The evaluation has been done both by simulation and in a real platform that consists of eight compute nodes divided in two racks. The cluster contains two nodes with Intel(R) Xeon(R) E5 with 8 cores and 256GB of RAM in one rack and six nodes with Intel(R) Xeon(R) E7 with 12 cores and 128GB of RAM in the other. The connection between nodes in the same rack is a 10 Gbps Ethernet, whilst the connection between racks is made through a 1 Gbps Ethernet. The I/O is based on Gluster parallel file system.

A. Scalability evaluation

The overhead of the LDMs depends on the sampling interval, which is user-defined. In our experiments we have tested the LDMs with a minimum sampling interval of one second, which produces less than 1.0% of CPU and consumption with a memory footprint of 160 KB. These overheads are obtained from the system log. It shows the time that a process is executing CPU operations and the total time that the process has been alive. Also, we can get the real size that a process occupies in memory, and the communication overhead is directly related to the size of a monitoring message and the sending time. Note that LDMs are executed in the same compute nodes as the applications, so this overhead should be keep as low as possible. The LDAs and LDS use more computational resources than the LDMs. Note that these components are supposed to be executed in nodes not exclusively used by the applications or with more resources than the standard compute nodes. These two modules are in charge of receiving, processing and re-transmitting the performance metrics provided by LDMs. For this reason the main source of overhead for them is not the CPU but the network traffic (especially if the sampling interval is reduced).

To estimate the scalability limit, we have simulated a cluster with OMNET++ [8] under different workloads. The simulated architecture corresponds to a simple deployment: n nodes (with one IO and network interface, and without GPU, that execute LDM functions) connected to a switch, and the switch connected to another node (that represents LDS). The connection between nodes and the switch is 1Gbps Ethernet. When we consider a particular computational-intensive configuration of LIMITLESS consisting of a sampling interval of one second and a single-thread implementation of the LDAs and LDS[2]. Under these circumstances in our experiments each LDA and LDS are able to be connected at the same time with 200 other components (either LDMs or LDAs). Consequently, one LDA in the first aggregation level is able to gather the metrics of 200 nodes, and one LDA in the second aggregation level could manage 40,000 nodes. Note that the packet size used for sending each sample ranges from 64 to 1024 bytes, according to the number of metrics collected by each LDM. Assuming that a sampling interval of 1 second and 40,000 nodes, the related bandwidth would range between 2.4Mbit/s and 39Mbit/s, much smaller than the typical bandwidth values of the network, like 10Gbits/s. Besides, note that the monitoring infrastructure could use the service network existing in many clusters producing no-overhead in the communication fabric.

In addition to this, we have done a performance test in a real cluster consisting of 50 nodes with this configuration per node: 2 Intel processors, 12 cores per processor, 256GB of RAM, and 44TB of global storage. The monitor was running for several months successfully with a sample period of 1 second.

[2]Note that in the latest version of LIMITLESS the LDS is multithreaded which considerably improves the performance.

B. *Smart analytic evaluation*

In our experiments, we have considered three different use cases. The first use corresponds to a parallel implementation Jacobi method written in MPI that alternates CPU, communication and I/O phases. Jacobi is executed using 8 processes with a matrix size of 15,000 entries. Given that the application is executed exclusively most of the time in one compute node, the duration of each phase is approximately constant during all the execution. Figure 4 shows the percentage of CPU in use during the use case execution. The spikes show a reduction in the CPU usage and corresponds to the I/O phases which are performed periodically. There is sharp increase in the CPU use around t=25,000 sec. that corresponds to another application running in the same compute node for a while.

The second use case corresponds to the simultaneous execution of two Jacobi applications that are exclusively executed in different nodes. Because of that, the duration of the CPU and communication phases is barely constant, but the I/O phase duration changes when both applications perform the I/O at the same time. In this case, there is an I/O interference that make that the I/O bandwidth would be shared by both applications, increasing the phase duration. Figure 5 shows the CPU use corresponding to this use case. The uses case uses 12 processes, consuming nearly 100% of the computer node's CPU. Now the distance between the spikes related to the I/O is not constant because of the I/O interference. The third use case corresponds to btio from NAS Parallel Benchmarks configured as class C. This benchmark alternates CPU and I/O phases. Figure 6 shows the CPU usage for this benchmark.

Table I shows the accuracy of each predictor for the three considered use cases using the performance metrics collected by LIMITLESS. This accuracy is expressed as a percentage of correct predictions. We used a tolerance value of 3% in each predictor. When we compare both executions of Jacobi method (use cases 1 and 2), the predictors are in general more accurate for the first use case because it has a more steady behaviour. Note that the amount of saved LDA network traffic is proportional to the predictor accuracy given that each correct prediction avoids sending one monitoring sample to the LDS. Regarding the memory usage, we do not include results for reasons of space, but these results have even better accuracy given that the memory use has less variations that the CPU use.

Note that the key idea of this approach is that both LDA and LDS execute the same predictor configured by the Analytic component. The predictors based on pattern matching and historical window have no input parameters, thus there is no transmission overhead. The parameters related to the neural network have a size of 28 KB whereas the ones related to the multi-variable correlation using the Nearest Neighbour Machine Learning algorithm have a size of 23 KB.

Table II shows the accuracy of each predictor for the first two use cases using the performance metrics collected by CLARISSE and FlexMPI and stored in ElasticSearch. It was not possible to integrate Btio in CLARISSE nor FlexMPi because it is written in Fortran. Note that these runtime use timestamps for recording the duration of the CPU, communication and I/O phases. This brings a more accurate representation of the application that enhances the modelling. As we can see in Table II the accuracy of the prediction algorithms is improved.

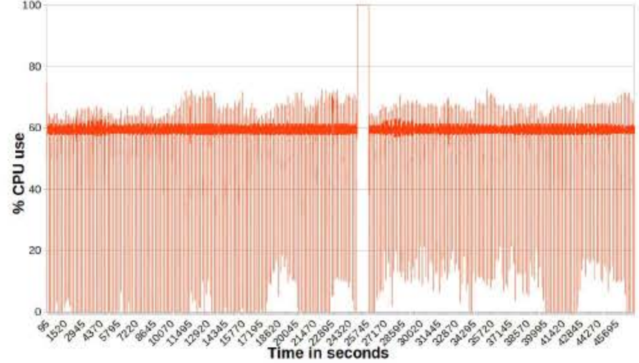

Fig. 4. CPU use of first use case, Jacobi method executed exclusively.

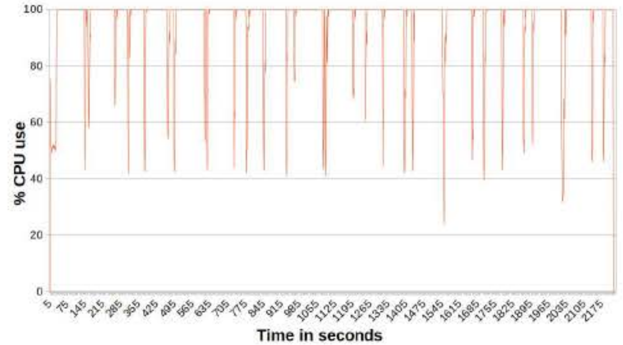

Fig. 5. CPU use of the second use case, Jacobi method executed with I/O interference.

We show only CPU information but the framework processes all metrics. We distinguish between two kinds of metrics: with high and low variability. For example, the first group includes CPU, IO, and Network, and the second one includes Memory and temperature. The first group metrics are harder to predict because of the variability. However, if the metrics has more constant values (like the second group), the predictor accuracy increases.

In a summary, all of these optimizations have an impact into the amount of packets sent by the monitor. Table III shows the overall traffic reduction related to the different

	Use case 1: Jacobi excl.	Use case 2: Jacobi interf.	Use case 3: btio
Pattern matching	25.2%	25.2%	24.7%
Historical window	61.5%	82.7%	50.0%
Neuronal networks	99.5%	93.3%	98.0%
Machine Learning	90.8%	90.5%	88.5%

TABLE I
ACCURACY OF THE DIFFERENT PREDICTION ALGORITHMS FOR THE THREE USE CASES EXPRESSED AS PERCENTAGE OF CORRECT PREDICTIONS.

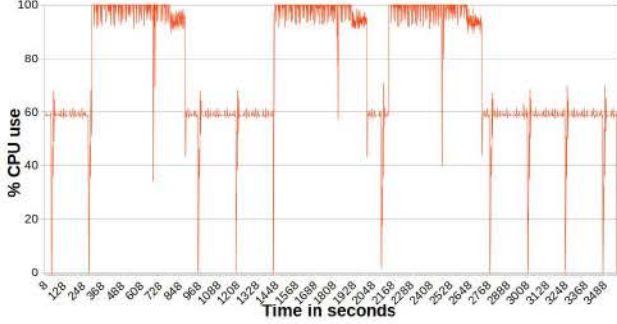

Fig. 6. CPU use of the third use case, btio code from NAS Parallel Benchmarks.

modelling algorithms. Note that those values includes the analysis of four collected metrics: CPU, IO, Memory and Network communications. These results are a bit worsen than the displayed in table I because of the extra metrics analysed and their variability.

VI. Related work

System monitoring for large-scale HPC platforms is a complex task, which becomes increasingly challenging as the scale and complexity of the infrastructure increases [4]. In the context of system monitors, we can find solutions like Ganglia [9], one of the most-used monitoring tools for HPC systems. One of its main features is that all the nodes receive information from the others. One drawback of this solution is that the fault tolerance mechanisms that use increase the communication and processing overhead and the amount of replicated information. Another similar tool is Collectd [10], which is a daemon that collects system and application performance metrics. Its main strength is that it has a low overhead. However, it only collects and stores the monitor metrics in the form of raw data. The user has to gather, manage and process the collected information. Another well-known framework for monitoring computer network is Nagios [11]. Nagios gathers a variety of information in clusters and is also able to get data from different services. The main drawback in Nagios is that it was not designed for HPC systems, so that, its performance and scalability in large-scale platforms is unclear. DiMMon [12] is a Distributed Modular Monitoring system that provides different paths to send the data, application profiling features, and dynamic reconfiguration of the modes. In addition to this, it has been adapted for decentralized edge clouds [13]. Besides these works, surveys of system monitoring tools based on collecting performance metrics are available in [14] and [15], but their application for HPC platforms is unclear.

Application performance monitoring in large scale systems is an important topic to enhance applications performance and to maximize system usage. Many applications exhibit a performance that changes among the time, thus some authors [16] have focused on making dynamic application profiling based on system monitoring. The usual technique is to collect system metrics and sending the information to a central component through a hierarchical model. Then, different algorithms are used to extract the knowledge about the application behaviour and to elaborate application profiles that are subsequently used to guide different components, like the scheduler. Other example of application monitoring based on system monitoring is [17], where the authors developed a system that parses the Slurm log file and extracts the information, generating reports for the users. LIKWID [18] is a framework that performs job or task monitoring. Its architecture is similar to LIMITLESS because its model is based on a monitor (Diamond), a non-SQL database (influxDB) and a visualizer (Grafana). The difference is that this solution is oriented to small and medium-sized clusters while LIMITLESS includes more scalable features, combines monitoring in two levels (system and application) and includes different features to reduce network traffic and perform application modeling. In a

	Use case 1: Jacobi excl.	Use case 2: Jacobi interf.
Pattern matching	48.5%	53.3.2%
Historical window	98.6%	98.7%
Neuronal networks	99.3%	99.5%
Machine Learning	98.7%	98.8%

TABLE II
ACCURACY OF THE DIFFERENT PREDICTION ALGORITHMS USING FLEXMPI AND CLARISSE AS INPUTS.

	Use case 1: Jacobi excl.	Use case 2: Jacobi interf.	Use case 3: btio
Pattern matching	24%	23%	4%
Historical window	60%	80%	50%
Neuronal networks	96%	92%	96%
Machine Learning	90%	90%	88%

TABLE III
PERCENTAGE OF NETWORK TRAFFIC SAVED OF ALL THE PREDICTION ALGORITHMS, INCLUDING CPU, IO, MEMORY AND NETWORK COLLECTED METRICS.

Features	Included	Future
System performance metrics	✓	
Process performance metrics	✓	
Scalability (> 200 nodes per aggregator)	✓	
Overhead in compute-nodes (< 0.1%)	✓	
Hierarchical model overhead (< 0.1%)	✓	
Fault tolerance (node level)	✓	
Fault tolerance (communication level)	✓	
Data redundancy (database)	✓	
Topological deployment	✓	
Auto-deployment based on topology		✓
Manage resources for energy reduction		✓
Node failure detection		✓
API to give data to external modules	✓	
Soft-real time visualization ($< 5secs$)	✓	
Smart monitor (take decisions based on ML&NN)	✓	
Application pathological behaviour detection	✓	
Monitor traffic optimization	✓	
Reporting		✓

TABLE IV
LIMITLESS FEATURES SUMMARY.

different area there are works that combine monitoring with machine-learning techniques. Yu et al. [19] present a paper that combines network monitoring with prediction techniques to design a cross-layer security algorithms for intrusion detection. Rashti et al. [20] uses prediction models in order to reduce the power consumption of a wireless sensor network. Tang et al. [21] proposed a similar work to predict a bridge monitoring data based on regressions using support vector machine. Xiaobing et al. [22] made a study of the historical data of a gas tunnel to predict the gas emission tendency based on monitor data. This work includes a detailed methodology description and an exhaustive evaluation of the prediction accuracy.

The monitor presented in this work (LIMITLESS) uses machine learning or neural networks to provide application modelling that is used for reducing the monitor traffic and enhancing the application scheduling. Table IV shows the features that are currently covered by our solution (LIMITLESS) such as system monitoring, dynamic scheduling based on monitoring, or machine learning and neural networks techniques for performance prediction. Note that we it is not possible to compare directly our proposed solutions to other related works because we have been unable to find a solution that includes a similar solution (based on in-transit processing) that the one presented in this paper.

VII. Conclusion

In this paper we introduce new features on LIMITLESS, a light-weight monitoring tool that was designed to monitor large-scale computing infrastructures. These features include topological-aware deployment, dynamic reconfiguration, insitu and in-transit processing for reducing the monitor traffic, performance modelling and event detection and notification. One of the main characteristics of LIMITLESS is its integration with other platform software components like the application scheduler, CLARISSE and FlexMPI runtimes. In this paper we present novel techniques that leverage this integration for reducing the monitoring traffic. By leveraging these features, LIMITLESS is able to reduce the monitoring traffic up to 85%, and to provide application profiles based on the detected performance. In this work we carried out different experiments showing that LIMITLESS exhibits a good scalability and is suitable for very large scale machines.

As a future work, we plan to monitor I/O interference and other application-related metrics for supporting multi-criteria application scheduling. We also plan to include new features to automatically detect the cluster topology with the objective of auto-deploy the monitor with an efficient layout. Finally, we want to extend this feature including more refined prediction algorithms for improving the in-transit processing algorithms.

References

[1] F. Isaila, J. Carretero, and R. Ross, "CLARISSE: A middleware for data-staging coordination and control on large-scale HPC platforms," in *16th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 346–355.

[2] G. Martín, D. E. Singh, M.-C. Marinescu, and J. Carretero, "Enhancing the performance of malleable mpi applications by using performance-aware dynamic reconfiguration," *Parallel Computing*, vol. 46, pp. 60 – 77, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819115000642

[3] C. Gormley and Z. Tong, *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine.* " O'Reilly Media, Inc.", 2015.

[4] A. Cascajo, D. E. Singh, and J. Carretero, "Performance-aware scheduling of parallel applications on non-dedicated clusters," *Electronics*, vol. 8, no. 9, p. 982, 2019.

[5] Y. Gupta, *Kibana essentials.* Packt Publishing Ltd, 2015.

[6] P. Cunningham and S. J. Delany, "k-nearest neighbour classifiers–," *arXiv preprint arXiv:2004.04523*, 2020.

[7] N. Wiebe, A. Kapoor, and K. M. Svore, "Quantum algorithms for nearest-neighbor methods for supervised and unsupervised learning," *Quantum Inf. Comput.*, vol. 15, pp. 316–356, 2015.

[8] A. Varga, "Omnet++," in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Gunes, and J. Gross, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 35–59.

[9] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[10] "collectd – The system statistics collection daemon." [Online]. Available: https://collectd.org/

[11] "Nagios - The Industry Standard In IT Infrastructure Monitoring," 2018. [Online]. Available: https://www.nagios.org/

[12] K. Stefanov, V. Voevodin, S. Zhumatiy, and V. Voevodin, "Dynamically Reconfigurable Distributed Modular Monitoring System for Supercomputers (DiMMon)," *Procedia Computer Science*, vol. 66, pp. 625–634, 2015.

[13] R. P. Centelles, M. Selimi, F. Freitag, and L. Navarro, "DIMON: Distributed monitoring system for decentralized edge clouds in guifi.net," in *Proceedings - 2019 IEEE 12th Conference on Service-Oriented Computing and Applications, SOCA 2019.* IEEE., 2019, pp. 1–8.

[14] C. Brown, B. Schwaller, N. Gauntt, B. Allan, and K. Davis, "Standardized Environment for Monitoring Heterogeneous Architectures," in *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, vol. 2019-September. Institute of Electrical and Electronics Engineers Inc., sep 2019.

[15] J. Eitzinger, T. Gruber, A. Afzal, T. Zeiser, and G. Wellein, "ClusterCockpit-A web application for job-specific performance monitoring," in *Proceedings - IEEE International Conference on Cluster Computing, ICCC.* IEEE Inc., 2019.

[16] A. Agelastos, B. Allan, J. Brandt, A. Gentile, S. Lefantzi, S. Monk, J. Ogden, M. Rajan, and J. Stevenson, "Continuous whole-system monitoring toward rapid understanding of production HPC applications and systems," *Parallel Computing*, vol. 58, pp. 90–106, 2016.

[17] J. Sperhac, B. D. Plessinger, J. T. Palmer, R. Chakraborty, G. Dean, M. Innus, R. Rathsam, N. Simakov, J. P. White, T. R. Furlani, S. M. Gallo, R. L. DeLeon, M. D. Jones, C. Cornelius, and A. Patra, "Federating XDMoD to Monitor Affiliated Computing Resources," in *Proceedings - IEEE International Conference on Cluster Computing, ICCC.* IEEE Inc., 2018, pp. 580–589.

[18] T. Rohl, J. Eitzinger, G. Hager, and G. Wellein, "LIKWID monitoring stack: A flexible framework enabling job specific performance monitoring for the masses," in *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, vol. 2017-September. Institute of Electrical and Electronics Engineers Inc., 2017, pp. 781–784.

[19] Y. Yu, L. Guo, J. Huang, F. Zhang, and Y. Zong, "A cross-layer security monitoring selection algorithm based on traffic prediction," *IEEE Access*, vol. 6, pp. 35 382–35 391, 2018.

[20] S. M. Rashti, M. Mollanoori, M. S. Nia, and N. M. Charkari, "A prediction-based algorithm for target tracking in wireless sensor networks," in *2009 International Conference on Ultra Modern Telecommunications and Workshops*, 2009.

[21] H. Tang, G. Tang, and L. Meng, "Prediction of the bridge monitoring data based on support vector machine," in *Proceedings - International Conference on Natural Computation*, vol. 2016-January. IEEE Computer Society, 2016, pp. 781–785.

[22] X. Kang and M. Xu, "Explore of monitoring data pattern prediction of gas tunnel," in *2011 International Conference on Remote Sensing, Environment and Transportation Engineering, RSETE 2011 - Proceedings*, 2011, pp. 4046–4049.