2025, **26** (4), 502–514. doi 10.26089/NumMet.v26r433

doi 10.26089/NumMet.v26r433

New approaches for automatic analysis of HPC application performance using the TASC software suite

Vladimir A. Matveev
Lomonosov Moscow State University, Research Computing Center, Moscow, Russia
ORCID: 0009-0005-3696-1297, e-mail: maculaali@yandex.ru

Alexander V. Setyaev
Lomonosov Moscow State University, Research Computing Center, Moscow, Russia
ORCID: 0009-0003-4030-4711, e-mail: alexsetyaev@gmail.com

Vadim V. Voevodin
Lomonosov Moscow State University, Research Computing Center, Moscow, Russia
Moscow Center of Fundamental and Applied Mathematics, Moscow, Russia
ORCID: 0000-0003-1897-1828, e-mail: vadim@parallel.ru

Abstract: This paper presents new automatic analysis approaches for identifying performance issues and useful properties in jobs running on a supercomputer. Methods for detecting problematic application classes, such as "hung" programs and jobs with underutilized nodes, are proposed. New assessments for automatic preliminary evaluation of GPU processors and memory usage efficiency are developed and tested as well. These approaches extend the functionality of the existing TASC software suite designed for conducting comprehensive analysis of usage quality of modern supercomputers.

Keywords: supercomputer, monitoring, performance analysis, HPC applications, supercomputer usage quality, application class, assessment system, resource usage efficiency.

Acknowledgements: The work was supported by the Ministry of Education and Science of the Russian Federation as part of the program of the Moscow Center for Fundamental and Applied Mathematics under Agreement No. 075–15–2025–345. The research was carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University.

For citation: V. A. Matveev, A. V. Setyaev, and V. V. Voevodin, "New approaches for automatic analysis of HPC application performance using the TASC software suite," Numerical Methods and Programming. **26** (4), 502–514 (2025). doi 10.26089/NumMet.v26r433.

2025, **26** (4), 502–514. doi 10.26089/NumMet.v26r433

Новые подходы для автоматического анализа производительности суперкомпьютерных программ с помощью программного комплекса TASC

В. А. Матвеев
Московский государственный университет имени М. В. Ломоносова,
Научно-исследовательский вычислительный центр,
Москва, Российская Федерация
ORCID: 0009-0005-3696-1297, e-mail: maculaali@yandex.ru

А. В. Сетяев
Московский государственный университет имени М. В. Ломоносова,
Научно-исследовательский вычислительный центр,
Москва, Российская Федерация
ORCID: 0009-0003-4030-4711, e-mail: alexsetyaev@gmail.com

В. В. Воеводин
Московский государственный университет имени М. В. Ломоносова,
Научно-исследовательский вычислительный центр,
Москва, Российская Федерация
Московский центр фундаментальной и прикладной математики, Москва, Российская Федерация
ORCID: 0000-0003-1897-1828, e-mail: vadim@parallel.ru

Аннотация: В данной статье представлены новые подходы к автоматическому анализу приложений, которые позволяют выявлять проблемы с производительностью или просто полезные свойства у любых заданий, выполняющихся на суперкомпьютере. Предложены методы для обнаружения проблемных классов приложений, таких как “зависшие” задания и задания со слабо загруженными узлами. Также разработаны и апробированы новые метрики для автоматической первичной оценки эффективности использования графических процессоров и памяти. Эти методы позволяют расширить функциональность существующего программного комплекса TASC, предназначенного для проведения разностороннего анализа качества работы современных суперкомпьютеров.

Ключевые слова: суперкомпьютер, мониторинг, анализ производительности, суперкомпьютерные приложения, качество работы суперкомпьютера, класс приложения, система оценок, эффективность использования ресурсов.

Благодарности: Исследование выполнено при поддержке Министерства образования и науки Российской Федерации в рамках программы Московского центра фундаментальной и прикладной математики, соглашение № 075–15–2025–345. Работа выполнена с использованием оборудования Центра коллективного пользования сверхвысокопроизводительными вычислительными ресурсами МГУ имени М. В. Ломоносова.

Для цитирования: Матвеев В.А., Сетяев А.В., Воеводин В.В. Новые подходы для автоматического анализа производительности суперкомпьютерных программ с помощью программного комплекса TASC // Вычислительные методы и программирование. 2025. **26**, № 4. 502–514. doi 10.26089/NumMet.v26r433.

1. Introduction. The performance analysis and optimization of HPC applications is an important and complex task that significantly impacts the quality of modern supercomputer functioning. Its importance arises from the fact that low performance of programs running on a supercomputer is one of the key factors that notably reduces the overall efficiency of the supercomputer work [1], and the idle time of supercomputer resources due to underutilization can be substantial.

The complexity is driven by several factors. Firstly, debugging, profiling, and especially optimizing parallel applications are often very challenging tasks, requiring considerable theoretical knowledge and practical skills.

2025, **26** (4), 502–514. doi 10.26089/NumMet.v26r433

Secondly, to develop a parallel program that runs efficiently on a modern supercomputer, it is necessary to take into account all peculiarities of hardware and system software, which are abundant in high-performance systems. Thirdly, supercomputer users are often unaware that their programs are working inefficiently, and it is often not trivial to detect this fact. For instance, the survey that we conducted among users of the MSU Supercomputing Center showed that more than a third of them are unaware of whether their applications have performance issues (and another third admit that problems exist and remain unresolved) [2].

The TASC software suite [3], previously developed in the MSU Research computing center, is primarily aimed at addressing this third global problem. TASC can automatically detect performance issues in HPC applications [4] and evaluate the efficiency of using supercomputer resources, as well as provide administrators with flexible report tools for analyzing various aspects of supercomputer functioning with the required level of detail [5].

This paper presents new approaches developed to extend the functionality of TASC. They include methods for the automatic detection of performance issues as well as useful properties of supercomputer applications, based on the monitoring data collected for the entire flow of running jobs. The paper also describes developments in expanding the assessment system [1], which allows evaluating (also automatically, based on monitoring data and for the entire job flow) the efficiency of using a specific type of resource by any supercomputing application. Previously, assessments were proposed and implemented for the following resource types: CPUs, memory, communication network, and I/O (input/output). This work extends the mentioned approach to graphics accelerators (GPUs) proposing assessments for GPU processors and GPU memory.

The main contribution of this paper is the development and description of new methods that allow automatically identifying important properties related to the performance and overall behavior of any application running on a supercomputer. Information about these properties is useful both for users (because they learn about issues in their applications and can eliminate them) and for supercomputer administrators (since it allows them to better understand the structure and problems of the job flow, which ultimately helps to improve the overall system efficiency).

The rest of the paper is organized as follows. Section 2 is devoted to methods for automatically detecting specific classes of jobs that provide useful information about the performance or behavior of supercomputer applications. Section 3 describes a proposed new approach for evaluating the efficiency of GPU processor and memory usage. Section 4 provides a brief summary of the results presented in this paper and future plans.

2. Automatic detection of job classes. Modern supercomputers constantly execute a large number of user jobs, but information about their properties is generally unavailable. And such knowledge would be useful and would help better adapt the work of the supercomputer center to the needs of users, as well as identify various issues in their applications [2]. In some cases, it is possible to identify job properties, and data from monitoring systems is usually used for this purpose. Monitoring system agents continuously run on the supercomputer's compute nodes and collect information on resource activity and usage efficiency (such as CPU load, cache miss rate, data volume transferred over the communication network, and so on).

In this work, we decided to use such data to implement methods for automatically detecting five classes of jobs (each class corresponds to the presence of a certain important property or issue):

- three classes of hung jobs:
  - (`job_hang`) the job initially ran in normal mode and then "hung" until its completion (i.e. its behavior corresponded to the behavior of programs that stopped at a certain point in execution and remained idle or continuously performed the same actions, without performing any useful work, but consuming computational resources);
  - (`cold_start`) the job's activity at start corresponded to a hung job, but then its behavior returned to normal;
  - (`stall`) the job initially operated normally, then for some time its behavior matched that of a hung job, and then its execution resumed as normal;
- (`idle_nodes`) the job was allocated several compute nodes, but it actively uses only a part of them;
- (`stable`) the job's behavior remains virtually unchanged throughout its execution.

Jobs of these classes are quite frequently encountered in practice, but they are not always easy to detect, which motivated the choice of these classes in this work.

2025, **26** (4), 502–514. doi 10.26089/NumMet.v26r433

There are several studies that focus on identifying different classes of supercomputer applications based on their behavioral or performance characteristics. For example, in the paper [6], supercomputer applications are grouped into classes according to resource usage patterns. However, the set of these classes differs significantly from the specified list of classes we are interested in, since the original goal of that paper was to automatically detect unwanted applications (such as cryptocurrency mining or password attack). In another paper [7], the authors identify inefficient applications by analyzing data for the first two minutes of job execution. The drawback of this approach is that the authors do not conduct a comparative analysis of the entire time series within job execution, but only examine the initial short period, which is not appropriate for our case (many supercomputer jobs run for hours and can significantly change their behavior over time). Furthermore, the problem statement in that work did not include identifying specific application classes, which is of interest in our case. The work [8] also considers among other aspects the issue of classifying supercomputer applications, but, as in the previous case, it did not address the issue of identifying the application classes of interest. In addition, the classes were determined based on the executable file name, which is not suitable for our purposes. There are many other studies that attempt to detect inefficient supercomputing applications (e.g., abnormally inefficient jobs [9]) or applications with peculiar properties (e.g., jobs with specific power consumption [10]), but they are even less suitable for solving the task posed in this paper. Therefore, it was decided to develop our own solution for this purpose.

Before moving on to a more detailed description of how these classes are defined, how their detection is implemented in practice, and when this might be useful, it is necessary to describe the data used for this analysis. In all cases, the input information was monitoring data collected by the DiMMon monitoring system [11] on the Lomonosov-2 supercomputer [12]. Note that the collected data is not specific to this monitoring system or supercomputer and can be gathered on most modern supercomputers using other means. The list of the collected characteristics is as follows (those characteristics that were immediately discarded as obviously inappropriate for the purposes of this work are not included here):

- CPU load (user and nice load) and load average;
- number of L1 and LLC cache misses per second;
- network data transfer (bytes/packets sent/received per second);
- parallel file system read and write rate (bytes/packets per second sent/received over the network for file system access);
- GPU utilization.

Monitoring data for each characteristic is collected once per minute. Initially, data is available for each node running the job, but in this study the average value over all nodes is used (except for `idle_nodes` class). Thus, each job is described by multivariate time series, where each time series represents minute-by-minute values of a certain performance characteristic during the job's execution, averaged over the participating nodes.

2.1. "Hung" job classes. The first three job classes are grouped together because they are closely related and all indicate that the program "hung" at some point in time. The causes of such behavior usually cannot be determined using monitoring data only, but it is worth noting that these reasons can vary significantly. For example, job hang can be caused by either incorrect operation of the application itself or by specific system software. The latter can occur, for instance, due to excessive load on the shared file system (which leads to a temporary halt of all applications accessing I/O at that moment) or because of the prolonged execution of service scripts launched immediately before or after the completion of a user job. It is often difficult not only to detect the causes but also the symptoms of such programs, because they can hang at different stages of execution resulting in significantly variations in their resource usage behavior. The main similarity between hung jobs studied in this paper is that their behavior changes sharply when the hang begins and becomes virtually constant during this time (though some fluctuations in values are almost always present).

Let us consider how to determine whether a job belongs to the `job_hang` class. The method is based on defining the variability of application performance characteristics collected by a monitoring system: relying on our experience it is assumed that the characteristic values change much less during a hang than during normal execution. The analysis is performed using a sliding window method, which enables considering large data intervals, but only local changes in the data are taken into account at the same time.

Let T_{start} and T_{end} be the beginning and the end of the program execution time interval, where the time interval corresponds to some number of the considered consecutive sliding windows. Let T be the minimum threshold value for the interval duration, T_i stands for the sliding window under consideration within the interval,

2025, **26** (4), 502–514. doi 10.26089/NumMet.v26r433

$Q1_{T_i}$ and $Q3_{T_i}$ are the boundaries of the first and third quartiles for the analyzed characteristic values in the window T_i, and θ be the heuristically selected threshold value. For each characteristic the following criterion is considered:

$$((T_{\text{end}} - T_{\text{start}}) > T) \ \wedge \ (\forall\, T_i \in (T_{\text{start}}, T_{\text{end}}) : \ (Q3_{T_i} - Q1_{T_i})/2 < \theta).$$

If the given criterion holds at the end of the job execution, and before that part there is at least one interval longer than T for which this criterion is not satisfied, then the flag of belonging to the `job_hang` class is raised for the considered characteristic. If this flag is raised for the majority of the characteristics under consideration, the given job is assumed as belonging to this class. In our case, T is equal to 60 (i.e. the length of the aforementioned interval must be not less than 60 minutes), the window size T_i is 20 minutes, and the value θ depends on the characteristic and was chosen to maximize the difference in the behavior of normal and hung jobs obtained in practice. In the case of `job_hang`, the following characteristics were selected: 1) L1 cache miss rate; 2) LLC cache miss rate; 3) the number of packets received over the communication network per second; 4) the volume of data in bytes sent over the network per second; 5) the average number of active processes on the node (i.e. load average).

To determine the best suited set of characteristics, all reasonable combinations of characteristics were tried, and the described set was selected as the most accurate (the accuracy assessment is shown below). The threshold values specified above were also selected heuristically, based on testing the method's accuracy for various threshold values. It should be noted that the set of most suitable characteristics, as well as the threshold values described above, may differ when porting this method to another computing system. We expect the accuracy of the method will remain sufficiently high in many cases with the selected parameters on other systems; however, to achieve higher values, it is recommended to rerun the process of selecting parameters and characteristics.

The methods for determining belonging to the `cold_start` and `stall` classes just slightly differ from the aforementioned method. Firstly, for these classes, the "hung" interval must be at the beginning and in the middle of the job execution, respectively. Secondly, other sets of characteristics were selected for them (the selection was carried out in the same way as for the `job_hang` class):

- `cold_start`: 1) L1 cache miss rate; 2) LLC cache miss rate; 3) CPU user load; 4) the amount of data in bytes received over the network per second; 5) the number of packets sent over the network per second;
- `stall`: 1) L1 cache miss rate; 2) LLC cache miss rate; 3) the amount of data in bytes received over the network per second; 4) the number of packets sent over the network per second; 5) the number of packets received over the communication network for reading from the shared file system per second; 6) the amount of data in bytes sent over the network for writing to the shared file system per second.

To evaluate the accuracy of class identification, an approbation was performed on the Lomonosov-2 supercomputer. A set of 4000 user jobs was selected and then analyzed using the three proposed methods for identifying hung jobs (note that this was a test set, which means that the parameter and characteristic selection was previously performed on another set). All jobs from these 4000, for which at least one of the methods identified belonging to a certain class, were selected, as well as approximately the same number of jobs that did not belong (according to the results of the methods) to any class. A total of 333 jobs were obtained. They were then manually labeled, which was used as a "ground truth".

The results of checking the correspondence between manual and automatic labeling of these jobs are presented in Tables 1 and 2, showing the overall accuracy values (the percentage ratio of correctly classified jobs to all studied jobs) as well as the confusion matrix for all three classes. In Table 2, "Prediction" relates to the corresponding method, while "Actual" refers to the manual labeling. It can be seen that in all cases the accuracy is high — over 92%. We also note that the number of false positive errors is very small — only 2 jobs for `job_hang`, 1 for `cold_start`, and 6 for `stall`. The decrease in the proportion of false positives was of particular

Table 1. Accuracy assessment for three "hung" classes

Class	Value, in %
`job_hang`	95.5
`cold_start`	98.2
`stall`	92.8

Table 2. Confusion matrix (`job_hang`/`cold_start`/`stall`)

		Prediction	
		True	False
Actual	True	45/24/63	2/1/6
	False	13/5/18	273/303/246

2025, **26** (4), 502–514. doi 10.26089/NumMet.v26r433

interest (more than the decrease in false negative errors), since belonging to, for example, the `job_hang` class implies a certain reaction — either notifying the user about it or taking automatic measures such as canceling the job. In this case, it is better to mistakenly ignore the existence of some issue and take no action than to mistakenly react to the existence of a non-existent issue.

2.2. "Idle_nodes" class. The `idle_nodes` class also detects issues in jobs, like the previous three classes. However, unlike the previous ones it detects cases in which a job running on several nodes actively utilizes only a few of them. Sometimes this occurs due to user error, when a job is designed for fewer processes/nodes than was allocated for it when it was added to the job queue. Another possible cause is improper distribution of processes/programs across nodes. For example, a user-defined job initially employs a script that distributes internal programs across nodes. If this script is configured incorrectly, all or almost all programs may run only on part of the nodes, while other nodes remain virtually unloaded.

It is worth noting that such nodes are not always completely idle, and that it can depend on the launch configuration and, thus, vary from run to run. This means that such cases cannot be detected solely based on absolute parameter values.

The approach outlined below was proposed to detect belonging to this class. The following four characteristics describing computational activity and resource utilization efficiency were selected for the analysis: 1) CPU user load; 2) L1 cache miss rate; 3) LLC cache miss rate; 4) GPU load. These characteristics were chosen based on expert assumptions about which of them are the most indicative for defining this class of jobs. Many other characteristics, for example, those related to network or file system usage, may not indicate activity even for a heavily utilized node simply because the corresponding supercomputer subsystems might not be used in the given job. Unlike determining hung jobs, in this case, data for each job is considered separately for each compute node involved in executing this job.

The following criterion was proposed for identifying a job to the considered class. For each of the four specified characteristics, two nodes were selected — one with the highest and one with the lowest average value for the corresponding metric over the job duration. Then, two conditions were checked:

1. The characteristic value on the maximum value node must exceed a predetermined threshold. This threshold is determined empirically based on an analysis of typical values for the corresponding characteristic under normal operating conditions. The thresholds are 1M misses per second for the first-level (L1) cache, 100K misses per second for the last-level (LLC) cache, 5% for the CPU load, and 50% for GPU load. In the future, it is planned to implement automatic methods for determining suitable thresholds based on the distribution of values for the characteristics; however thresholds are currently selected manually.
2. The characteristic value on this node must be at least twice the value on the minimum value node.

If the specified conditions are satisfied for at least half of the considered characteristics (i.e. for at least two out of four), the job is identifying as belonging to this class.

The method was tested on real jobs running on the Lomonosov-2 supercomputer. Three jobs of this class were found in a sample of 203 jobs. The method identified all jobs correctly. Although such jobs are relatively rare, it is important to identify them, since detection and elimination of these jobs enables more efficient use of supercomputer resources.

In the future, it is planned to expand the test dataset to conduct a more detailed evaluation of the method quality and assess the frequency of occurrence of such jobs in practice.

2.3. "Stable" class. The `stable` class, unlike the previous classes, does not indicate the presence of any issues in the application being analyzed. In this case, we are interested in jobs whose behavior remains virtually unchanged during execution, meaning the performance characteristics stay approximately the same. This class is primarily useful for supercomputer administrators and analysts, as applications within this class are very accurately described by integral characteristics (such as average and maximum CPU load over the entire execution period), which are most often used for initial application analysis. In other words, if a job belongs to this class, we can be confident that the integral values accurately describe its behavior. In practice, such jobs are commonly observed, making their detection a relevant task.

We will consider a job to belong to this class if the curves, showing the values of a selected set of job performance characteristics during its execution, each can be represented with sufficient accuracy by a horizontal line. However, it should be taken into account that small fluctuations in values are almost always observed (due to both small differences in the behavior of the application itself and changes in the external software and hardware environment), therefore some deviation from the straight line is evident in all applications.

2025, **26** (4), 502–514. doi 10.26089/NumMet.v26r433

The method for verifying the belonging of a job to this class consists of two main steps. First, one needs to approximate the characteristic values collected during application execution (approximation is performed separately for each characteristic), i.e. approximate a one-dimensional time series. Two options were considered: linear regression methods and using the mean value (since we are looking for jobs whose behavior is virtually constant, more complex approximation options are not needed in this case). Linear regression is usually much more accurate; however, in this case, mainly because the characteristic values for stable jobs are virtually constant, the accuracy of the method in both cases was approximately the same, but calculating the mean value is much faster, so this option was chosen.

Next, one needs to choose a metric to evaluate the difference between the actual values and the approximating line. Since the mean value was selected as the approximation method, a simple metric is suitable in this case, which at each point x of the time series for some characteristic calculates the deviation R of the actual data from the "ideal stable job" (the approximating horizontal line): $R = |y_{\text{fact}}(x) - y_{\text{mean}}|/y_{\text{mean}}$, where $y_{\text{fact}}(x)$ is the real value of some characteristic at point x, and y_{mean} is the mean value for this characteristic.

Further, for each characteristic, we require that R should not exceed the value of 0.1 for 95% of time, and the R value averaged over the entire job execution time should not exceed 0.05. These conditions admit the rare occurrence of actual characteristic values that significantly deviate from the approximating mean line (which can happen, for example, due to failures in the monitoring system or the presence of notable local environmental influences), but on average this deviation should be small. These thresholds were selected heuristically based on an analysis of the method's performance on the Lomonosov-2 supercomputer, but they can easily be adapted for execution on other computing systems.

A crucial question is what characteristics should be used to check this condition. The following approach was chosen for these goals. The stability of the entire job is assessed by voting with selection of characteristics divided into primary and secondary groups. The primary group includes the following characteristics: L1 cache miss rate, LLC cache miss rate, CPU user load, average number of active processes per node (load average), and GPU load. The secondary group involves four characteristics related to network usage: the amount of bytes/packets sent/received over the network per second. A job is considered stable if all characteristics in the primary group and at least half in the secondary group are stable (i.e. the above condition is met for them). This list of characteristics was selected heuristically as the one that demonstrated the best accuracy after using all reasonable combinations of characteristics. The division into two groups was proposed based on an expert opinion regarding which characteristics may be more or less important for defining a given class.

This method was tested on real jobs from the Lomonosov-2 supercomputer. The model's accuracy on a sample of 101 jobs was 0.97, while 25 stable jobs were found and 3 stable jobs were classified as unstable (false negative errors). None of the unstable jobs were identified by the program as stable (false positive errors).

2.4. Integration with TASC. As part of this work, a standalone software solution was developed in Python that implements methods for automatically identifying the aforementioned job classes based on data from the monitoring system. This solution was integrated into the existing Lomonosov-2 supercomputer performance analysis infrastructure.

This solution automatically checks all jobs running on the supercomputer for their membership in the selected classes and reports these results to the TASC system. The checks for different classes are performed independently and are easily configurable with the ability to simply add methods for detecting new classes. The solution runs in two Docker containers (one for the Python solution, another for the local PostgreSQL DB), receiving initial data from TASC and sending back the results of the class checks upon the request from TASC. Thus, the overall operation scheme of this solution consists of the following steps: data collection → storing data → periodic checking for class membership → saving the result → sending the result to TASC upon request.

The proposed solution runs at a preset frequency — by default, every 30 minutes (this interval can be easily adjusted). At each launch, it analyzes all jobs completed since the previous activation. The approach itself allows for the analysis of any jobs, both completed and still running. But at this moment, it was decided to consider only completed jobs to evaluate the solution. In the future, consideration of running jobs will be enabled to allow for a more rapid response to emerging issues.

The proposed implementation is easily portable to other supercomputers, but may require tuning of supercomputer-specific parameters (threshold values for class assignment, as well as the list of the most suitable characteristics).

For the Lomonosov-2 supercomputer, the developed implementation of class detection methods fully processes one job in an average of 3-5 minutes (experiments were conducted under the following conditions: one

2025, **26** (4), 502–514. doi 10.26089/NumMet.v26r433

Intel Xeon Gold 6238R processor core, 4 GB of RAM), most of which is spent working with databases (these DB contain detailed information on approximately four months of the supercomputer's operation and are therefore quite large). This performance is far enough for processing the entire Lomonosov-2 job flow.

3. Automatic assessment of GPU utilization efficiency. As mentioned earlier, users are often unaware that their applications running on a supercomputer are experiencing performance issues. This is a serious problem that significantly impacts the overall performance of the supercomputer. The approach proposed in the previous section is based on the automatic detection of specific application properties. However, there are various performance issues that need to be detected and corrected, and it is not possible to develop individual detection methods for each of them. Therefore, another approach is also relevant: one can evaluate the efficiency of using different computational resources, and if low usage efficiency is detected, this indicates the likely presence of issues related to this resource. Then, these issues can be further investigated using existing analysis tools such as profilers or debugging tools.

Previously, an assessment system [1] was developed in the MSU Research Computing Center that automatically collects efficiency scores for all jobs running on a supercomputer for four types of resources:

- CPU;
- memory subsystem;
- communication network;
- I/O (file system usage).

Next, we will first briefly outline the existing system and then describe a new approach that extends this system to graphics accelerators.

3.1. General assessment system. The goal of the previously developed assessment system is to provide a convenient and accurate initial method for evaluating the efficiency of resource usage by user applications. The assessment should be performed automatically for all running applications. For this aim it was decided to build such a system based on the data from the monitoring system. Each resource type is assessed separately and independently.

It is worth noting that supercomputer centers often use existing metrics for such purposes, for instance, CPU user load, cache miss rate, number of the performed read operations etc. Although these metrics are indeed useful (as they help evaluate the utilization of a given hardware component), they are often poorly suited for assessing resource efficiency.

The aforementioned assessment system was developed precisely for this purpose. In this study, the efficiency of using a certain resource measures how much working with this resource interferes with useful computations.

The analysis for different resource types is built differently. Assessments for CPU and the memory subsystem are based on metrics proposed in the Top-Down approach [13] developed by Intel. These assessments are calculated using the values of processor hardware counters. Assessments for the communication network and I/O were proposed based on a set of rules developed within the TASC framework, each of which identifies the presence of one specific performance issue in application [4]. Thus, different types of source data are used, but all aforementioned information sources are collected using a single DiMMon monitoring system (there is no strict binding to a specific monitoring system, and this solution will work with any other system that provides necessary monitoring data).

These assessments were implemented and tested on the Lomonosov-2 supercomputer. The practice has shown that they can be useful in different cases, for instance, when identifying issues commonly used metrics typically fail to reveal. For example, there were found two supercomputer users whose jobs had very high values for the $score_{\text{cpu}}$ metric, which reflects the degree of CPU usage inefficiency (the higher the value, the less efficiently processors are used). At the same time, the average CPU user load and the load average (widely used metrics for assessing CPU utilization) showed very high values, i.e. they did not indicate any performance issues like the rest of the collected monitoring data. A detailed examination of jobs of these users revealed that their jobs were sent to two nodes that were overheating at that time (note that temperature sensors for these nodes were not available, so no information about this issue was available). As a result, the programs running on these nodes worked noticeably slower than usual. Thus, the assessments from the proposed system allowed us to identify performance issues that were not detected by other methods.

2025, **26** (4), 502–514. doi 10.26089/NumMet.v26r433

3.2. Proposed formulas for GPU assessment. The aim of this work direction was to expand the assessment system by proposing and testing new methods for evaluating GPU performance. Graphics devices have their own processors and memory, so it was necessary to develop formulas for two scores — $score_{\text{gpu}}$ and $score_{\text{gpumem}}$, respectively.

The first question to be addressed was which existing technology for collecting GPU performance information should be used in this case. We did not consider software tools that require code instrumentation (such as NVIDIA Nsight Compute [14], nvprof or PC Sampling [15]), since we need to collect information about all running applications without interfering with their code or attaching to their processes. Therefore, we considered the following options, which do not require instrumentation and are available for collection using monitoring systems:

- NVML [16] and DCGM [17] built on its basis;
- PM Sampling [18] via CUPTI library.

NVML (like DCGM) is a lightweight tool that does not require a process attachment to obtain GPU metrics. However, this solution does not allow getting the characteristics needed for our goals, as the set of metrics available in NVML is limited. The only found alternative for assessing $score_{\text{gpu}}$ using NVML or DCGM is the *gpu_utilization* metric, which shows the fraction of time the GPU was executing at least one compute core. This metric is useful for assessing GPU utilization, but, unlike $score_{\text{gpu}}$ proposed below, it only indirectly indicates GPU efficiency usage and can be misleading in some specific cases.

Therefore, PM Sampling was chosen. This method allows collecting low-level data on GPU performance for external applications without modifying them. PM Sampling itself was only introduced in CUDA 12.6, released in August 2024.

During the analysis of possible formulations for GPU efficiency assessment the existing approaches were studied. Two works stand out as being the most closely related. In an earlier paper by the authors from the MSU Research Computing Center [1] the initial formulas for these two assessments were proposed. Another paper [19] presents formulas for microarchitectural analysis of GPU performance. However, a review of these works, as well as our own research revealed that not all metrics needed for calculating these formulas are currently available in PM Sampling. Therefore, it was decided to develop two formulas in each case: the first one (*theory*) is more accurate but cannot currently be collected by PM Sampling, while the second one (*practice*) is the most suitable of those that can be collected by PM Sampling. We decided to develop a theoretical formula as well because: firstly, it was interesting to understand how significantly the "ideal" formula differs from the one available in practice; secondly, in future versions of PM Sampling (as well as if another method of collecting data or greater capabilities in new GPUs appear in the future), it is quite possible that the ability to collect them will be present. It should be noted that within this study we examined a significantly larger number of different formula variants than those presented here. However, for the sake of brevity, only the final formula versions are shown in this paper.

Let us start by examining the formulas for $score_{\text{gpu}}$. In our opinion, the most appropriate formula for evaluating the efficiency of GPU usage within the proposed approach is the following (note that all formulas are defined in such a way that "0" means no problem at all and "100" corresponds to maximum problem level):

$$score_{\text{gpu}}^{\text{theory}} = 100 \cdot (1 - IPC_{\text{reported}} \cdot Efficiency_{\text{warp}} / IPC_{\text{max}}), \tag{1}$$

where IPC_{reported} is the average number of instructions executed per cycle on a single Streaming Multiprocessor (SM), $Efficiency_{\text{warp}}$ is the percentage of active threads per warp (those actually executing useful instructions) relative to the maximum possible number of threads running simultaneously (note that this differs from the number of active warps), and IPC_{max} is the maximum theoretically achievable number of instructions per cycle. This formula is based on the one proposed in the paper [19], and it estimates the ratio of the actually achieved number of instructions per cycle to the maximum possible, excluding "useless" instructions (those that were either recalculated or discarded due to branch misprediction). However, this formula is only theoretical in our case, since it cannot be collected using PM Sampling. For practical usage, we propose the following variant:

$$score_{\text{gpu}}^{\text{practice}} = 100 \cdot (1 - (warps\_active\_sup / max\_processing\_warps\_per\_sm) \cdot (sm\_\_cycles\_active / gpc\_\_cycles\_elapsed)), \tag{2}$$

where *max_processing_warps_per_sm* is the maximum number of simultaneously executing warps on SM per cycle (it is a theoretical constant parameter described in the specification of a particular GPU),

2025, **26** (4), 502–514. doi 10.26089/NumMet.v26r433

sm__cycles_active is the average number of GPU cycles over all SMs during which each SM was active, *gpc__cycles_elapsed* is the maximum number of GPC (Graphics Processing Cluster, which includes all SM) cycles that have elapsed per second, which is the upper bound on the number of active cycles, and *warps_active_sup* is equal to $\min\{max\_processing\_warps\_per\_sm, sm\_\_warps\_active\}$, where *sm__warps_active* is the average number of active warps over all SMs.

This formula estimates both how many resident warps were executed on average and how much of the execution time the GPU (in particular, streaming multiprocessors) was active. It is worth noting that this formula does not estimate the "efficiency" of the computations, which the theoretical formula above can provide to some extent, but that is the price for the ability to collect data in practice.

In the case of $score_{\text{gpumem}}$, the situation is generally similar. The formula based on the one proposed in the paper [19] was chosen as the best *theory* option among all others. However, it also could not be collected in our case, so we proposed our own *practice* version of the assessment which can be collected using PM Sampling. Formulas (3) and (4) describe these variants.

$$score^{\text{theory}}_{\text{gpumem}} = \sum_{<m>\in M} smsp\_\_warps\_issue\_stalled\_<m>\_per\_warp\_active.pct, \tag{3}$$

where $M = \{drain, imc\_miss, lg\_throttle, long\_scoreboard, membar, mio\_throttle, short\_scoreboard, tex\_throttle\}$,

$$score^{\text{practice}}_{\text{gpumem}} = 100 \cdot dram\_\_cycles\_active/(dram\_\_cycles\_active + sm\_\_cycles\_active). \tag{4}$$

In the case of $score^{\text{theory}}_{\text{gpumem}}$, each summarized metric yields the percentage of warps that were stalled due to some memory-related reason, and the formula lists all available metrics of that type. Here is a brief description of what these metrics measure (the metric's notation used in the formula above is given in parentheses):

- waiting for all memory-related instructions to complete (*drain*);
- waiting due to a cache miss (*imc_miss*);
- waiting for free space in the L1 instruction queue (*lg_throttle*);
- waiting for operations with variable latency (*long_scoreboard*);
- waiting on memory barriers (*membar*);
- waiting for free space in the memory instruction queue (*mio_throttle*);
- waiting for data from memory (*short_scoreboard*);
- waiting for free space in the texture cache instruction queue (*tex_throttle*).

In the $score^{\text{practice}}_{\text{gpumem}}$ formula, *dram__cycles_active* represents the number of GPU cycles in which DRAM was active (i.e. cycles in which memory operations were performed). This formula is significantly simpler than the *theory* one and roughly estimates the fraction of time during program execution when memory was in use. Note that this formula does not consider properly the cases when memory usage overlaps with calculations, since such cycles contribute both in *dram__cycles_active* and *sm__cycles_active*. We are aware that this assessment provides a mediocre representation of memory efficiency, but at the moment there is no more suitable version of this formula that could be collected in practice using PM Sampling has not been devised. We plan to improve this formula in the future as the capabilities of this collection method or new graphics accelerators increase.

3.3. Accuracy and overhead estimation. As mentioned earlier, formulas (1) and (3) are theoretical and cannot be tested in practice. However, this can and should be done for formulas (2) and (4), which are the subject of this section.

A software tool was implemented that collects the required metrics using PM Sampling. This tool confirmed in practice that collecting the metrics needed to calculate the proposed assessments does not require instrumentation or modification of the source code of user applications, nor does it require superuser privileges, making it possible to apply this approach within the overall assessment system (which uses the input data collected by the monitoring system).

First, it was necessary to verify the accuracy of the values collected in this way. To do this, it was decided to compare the data produced by this tool with that obtained using NVIDIA Nsight Compute (NCU), one of the most widely utilized solutions for profiling GPU applications. Experiments were carried out on GPU versions [20] of the NPB benchmark suite [21].

The evaluation was conducted in the following way. Using NCU, we collected instrumentation data for the same metrics that were utilized in the formulas, but they were taken as absolute values rather than values

2025, **26** (4), 502–514. doi 10.26089/NumMet.v26r433

relative to program execution time (per_second), as it is done within our tool. The NPB benchmarks are deterministic and perform the same operations each run. This means that execution time and used resources should vary minimally between runs. Therefore, if we take the absolute number of, for example, active SM cycles and divide this number by the average program execution time, we expect to obtain the same value as that returned by PM Sampling. After that we can calculate the same values as in the formulas above. As can be seen from Table 3, the values of scores based on data from NCU and from PM Sampling differ by no more than 4%.

Table 3. The difference (in percent) between the values of scores obtained in two ways

	BT		FT		IS		MG		SP	
	B	C	B	C	B	C	B	C	B	C
$score^{\text{practice}}_{\text{gpu}}$	2.94	3.31	3.98	2.55	0.61	1.68	1.96	0.67	3.85	3.82
$score^{\text{practice}}_{\text{gpumem}}$	3.77	2.25	3.66	3.44	1.97	2.09	0.67	3.92	3.88	3.90

Thus, in most cases, the metric collection accuracy is high and sufficient for correctly determining values according to the proposed formulas.

Next, it was necessary to evaluate the overhead introduced by the proposed tool. This issue is crucial, as the developed formulas are intended for the use within a monitoring system running on supercomputer nodes in parallel with user applications. In this case, it is necessary to minimize the impact of the data collection process on the applications themselves.

To evaluate the overhead, experiments were conducted on a server with A100 40GB GPU. Different GPU versions of the NPB benchmarks were run both with and without the assessment tool running in parallel. The execution times of two variants for each test were then compared, allowing us to estimate how much the data collection slows down the user application (see Table 4). All runs were repeated at least three times, and the average values are presented below (in all cases, the resulting slowdown or speedup was statistically significant). For faster tests, multiple iterations were used to ensure execution times were closer to other tests.

Table 4. Comparison of NPB test execution times with and without the assessment tool running in parallel

NPB Test	Class	Number of iterations	Time without PM_S, s	Time with PM_S, s	Slowdown, in %
SP	D	1	65.26	65.56	0.46
BT	C	5	59.26	59.88	1.04
FT	C	10	27.09	27.33	0.90
CG	C	10	106.40	108.00	1.50
MG	C	10	31.17	31.98	2.60
SP	C	10	27.72	27.86	0.50
LU	D	1	73.49	73.96	0.64
LU	C	5	34.25	33.96	−0.86
LU	B	10	45.41	44.32	−2.39
LU	A	20	45.48	43.20	−5.03

It can be seen from the “Slowdown, in %” column that only the MG test shows slowdown greater than 1%; in all other cases, it is less than 1%. This is a very good result indicating that the proposed approach to metric collection is applicable in practice in terms of the amount of overhead it introduces. For one of the tests (the LU test), we evaluated the overhead for different input data sizes. It may be observed that in this case, the overhead not only decreases with the input data size but becomes negative. In other words, when the data collection tool was running in parallel, the test execution time actually decreased. The exact reason for this behavior is still unknown; presumably, some system processes in GPU operate in a different mode when PM Sampling is enabled, resulting in a slight speedup (the shorter the test itself, the more speedup). In any case, the main conclusion that the overhead of collecting metrics with PM Sampling remains low is correct.

A separate aspect of interest was the impact of data collection frequency on the amount of overhead and the accuracy of the collected values for different metrics. By default, a data collection rate was one time in every 50 million clock cycles. The execution times of the NPB CG class A, as well as CG, FT, IS, MG and SP

2025, **26** (4), 502–514. doi 10.26089/NumMet.v26r433

class C benchmarks, were compared at the following data collection rates: 1 collection for every 35, 50, 100, 500 and 1000 million clock cycles. The results showed that the difference in metric values does not exceed 3% (and most often less than 1%), which is an entirely acceptable result for the purposes of this study. Changing the data collection rate also has an insignificant effect on program execution time. Moreover, the difference is statistically insignificant in all cases.

Thus, it was demonstrated that the proposed approach for obtaining assessments for determining the efficiency of GPU processor and memory usage is applicable in practice, since it is sufficiently accurate and does not introduce significant overhead.

4. Conclusions. In this paper, we propose new methods for automatic identification of useful performance-related information about supercomputer applications. These methods are based on the analysis of monitoring data collected for all jobs running on a supercomputer. This paper presents two directions of research. Within the first one, methods for identifying applications with specific properties are proposed. Specifically, methods were developed for detecting three types of hung jobs, stable jobs (whose behavior can be accurately described by integral performance characteristics, which is important for supercomputer administrators), and multi-node programs that actively utilize only a subset of the allocated nodes. The second research direction involves expanding the existing assessment system aimed to automatically conduct a preliminary evaluation of the usage efficiency for different resource types. This paper proposes two new assessments: one for GPU processors and another for GPU memory. Formulas for calculating them were developed, and the initial practical approbation of them was conducted.

Future plans include conducting a larger-scale evaluation of the proposed methods, as well as continuing to explore the most suitable formulas for the assessments. We also plan to integrate a software tool for calculating assessments into the existing monitoring system, which will allow for the automatic constant collection and analysis of the proposed assessments for all jobs running on the Lomonosov-2 supercomputer.

References

1. V. V. Voevodin, D. I. Shaikhislamov, and D. A. Nikitenko, "How to Assess the Quality of Supercomputer Resource Usage," Supercomputing Frontiers and Innovations **9** (3), 4–18 (2022). doi 10.14529/jsfi220301.
2. D. A. Nikitenko, P. A. Shvets, and V. V. Voevodin, "Why do Users Need to Take Care of Their HPC Applications Efficiency?" Lobachevskii Journal of Mathematics **41** (8), 1521–1532 (2020). doi 10.1134/s1995080220080132.
3. V. V. Voevodin, D. I. Shaikhislamov, and V. A. Serov, "TASC Software for HPC Performance Analysis: Current State and Latest Developments," Bulletin of the South Ural State University Series Computational Mathematics and Software Engineering **13** (3), 61–78 (2024). doi 10.14529/cmse240304.
4. P. Shvets, V. Voevodin, and S. Zhumatiy, "Primary Automatic Analysis of the Entire Flow of Supercomputer Applications," in *Proceedings of the 4th Ural Workshop on Parallel, Distributed, and Cloud Computing for Young Scientists, Yekaterinburg, Russia, November 15, 2018* CEUR Workshop Proceedings, Vol. 2281, pp. 20-–32.
5. P. A. Shvets, and V. V. Voevodin, " 'Endless' Workload Analysis of Large-Scale Supercomputers," Lobachevskii Journal of Mathematics **42** (1), 184—194 (2021). doi 10.1134/s1995080221010236.
6. E. Ates, O. Tuncer, A. Turk, et al., "Taxonomist: Application Detection Through Rich Monitoring Data," in *Proceedings of Euro-Par 2018: Parallel Processing, Turin, Italy, August 27–31, 2018* Lecture Notes in Computer Science Vol. 11014, pp. 92—105. doi 10.1007/978-3-319-96983-1_7.
7. T. Jakobsche, N. Lachiche, A. Cavelan, and F. M. Ciorba, "An Execution Fingerprint Dictionary for HPC Application Recognition," in *Proceedings of 2021 IEEE International Conference on Cluster Computing (CLUSTER), Portland, USA, September 7–10, 2021* IEEE Press, New York, 2021, pp. 604–608. doi 10.1109/Cluster48925.2021.00092.
8. R. D. Lewis, Z. Liu, R. Kettimuthu, and M. E. Papka, "Log-Based Identification, Classification, and Behavior Prediction of HPC Applications," in *Proceedings of HPCSYSPROS'20: HPC System Professionals Workshop, Atlanta, GA, USA, November 11–13, 2020* ACM, New York, 2020, pp. 1–7.
9. A. Bezrukov, M. Kokarev, D. Shaykhislamov, V. Voevodin, S. Zhumatiy, "Machine Learning Techniques for Detecting Supercomputer Applications with Abnormal Behavior," in *Proceedings of 12th Int. Conference on Parallel Computational Technologies (PCT 2018), Rostov-on-Don, Russia, April 2—6, 2018* Communications in Computer and Information Science 2018. Vol. 910, pp. 31—46. doi 10.1007/978-3-319-99673-8_3.
10. K. Yamamoto, Y. Tsujita, and A. Uno, "Classifying Jobs and Predicting Applications in HPC Systems," in *Proceedings of ISC on High Performance Computing, Frankfurt, Germany, June 24–28, 2018*, Lecture Notes in Computer Science Vol. 10876, pp. 81–99. doi 10.1007/978-3-319-92040-5_5.

2025, **26** (4), 502–514. doi 10.26089/NumMet.v26r433

11. K. Stefanov, Vl. Voevodin, S. Zhumatiy, and V. Voevodin, "Dynamically Reconfigurable Distributed Modular Monitoring System for Supercomputers (DiMMon)," Procedia Computer Science **66**, 625—634 (2015). doi 10.1016/j.procs.2015.11.071. https://doi.org/10.1016/j.procs.2015.11.071 Cited November 14, 2025.
12. Vl. Voevodin, A. Antonov, D. Nikitenko, et al., "Supercomputer Lomonosov-2: Large Scale, Deep Monitoring and Fine Analytics for the User Community," Supercomputing Frontiers and Innovations **6** (2), 4—11 (2019). doi 10.14529/jsfi190201.
13. Top-down Microarchitecture Analysis Method. https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023-0/top-down-microarchitecture-analysis-method.html. Cited November 14, 2025.
14. NVIDIA Nsight Compute Documentation. https://docs.nvidia.com/nsight-compute/. Cited November 14, 2025.
15. Description of PC Sampling in CUPTI Library. https://docs.nvidia.com/cupti/main/main.html#cupti-pc-sampling-api. Cited November 14, 2025.
16. NVIDIA Management Library (NVML) homepage. https://developer.nvidia.com/management-library-nvml. Cited November 14, 2025.
17. NVIDIA Data Center GPU Manager (DCGM) homepage. https://developer.nvidia.com/dcgm. Cited November 14, 2025.
18. Description of PM Sampling in CUPTI Library. https://docs.nvidia.com/cupti/main/main.html#cupti-pm-sampling-api. Cited November 14, 2025.
19. A. Saiz, P. Prieto, P. Abad, et al., "Top-Down Performance Profiling on NVIDIA's GPUs," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS), Lyon, France, May 30—June 3, 2022* IEEE Press, New York, 2022, pp. 179–189. doi 10.1109/IPDPS53621.2022.00026.
20. NAS Parallel Benchmarks for GPUs. https://github.com/GMAP/NPB-GPU. Cited November 14, 2025.
21. D. Bailey, T. Harris, W. Saphir, et al., "The NAS Parallel Benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center **156** (1995).

Received	*Accepted*	*Published*
October 25, 2025	*November 11, 2025*	*November 24, 2025*

Information about the authors

Vladimir A. Matveev — Technician; Lomonosov Moscow State University, Research Computing Center, Leninskie Gory, 1, building 4, 119234, Moscow, Russia.

Alexander V. Setyaev — Programmer; Lomonosov Moscow State University, Research Computing Center, Leninskie Gory, 1, building 4, 119234, Moscow, Russia.

Vadim V. Voevodin — Ph.D., Head of Laboratory; 1) Lomonosov Moscow State University, Research Computing Center, Leninskie Gory, 1, building 4, 119234, Moscow, Russia; 2) Moscow Center of Fundamental and Applied Mathematics, Leninskie Gory, 1, 119991, Moscow, Russia.