EPJ Web of Conferences **295**, 04005 (2024) https://doi.org/10.1051/epjconf/202429504005
CHEP 2023

Multicore workflow characterisation methodology for payloads running in the ALICE Grid

Marta Bertran Ferrer[1,*], *Costin* Grigoras[1,**], and *Rosa M.* Badia[2,***]

[1]CERN, Esplanade des Particules 1, 1211 Geneva 23, Switzerland

[2]Barcelona Supercomputing Center, Plaça Eusebi Güell, 1-3, 08034 Barcelona, Spain

Abstract. For LHC Run3 the ALICE experiment software stack has been completely refactored, incorporating support for multicore job execution. Whereas in both LHC Run 1 and 2 the Grid jobs were single-process and made use of a single CPU core, the new multicore jobs spawn multiple processes and threads within the payload. Some of these multicore jobs deploy a high amount of short-lived processes, in the order of more than a dozen per second. The overhead of starting so many processes impacts the overall CPU utilization of the payloads, in particular its *System* component. Furthermore, the short-lived processes were not correctly accounted for by the monitoring system of the experiment. This paper presents the developed new methodology for supervising the payload execution.

We also present a black box analysis of the new multicore experiment software framework tracing the used resources and system function calls issued by MonteCarlo simulation jobs. Multiple sources of overhead in the lifecycle of processes and threads have thus been identified. This paper describes how the source of each was traced and what solutions were implemented to address them. These improvements have impacted the resource consumption and the overall turnaround time of these payloads with a notable 35% reduction in execution time for a reference production job. We also introduce how this methodology will be used to further improve the efficiency of our experiment software and what other optimization venues are currently under research.

1 Introduction

The ALICE experiment software stack has evolved considerably from LHC Run2 to Run3. All of its components have been updated to accommodate the requirements of working with the higher volume of data being extracted from the detector per unit of time. Working with a larger amount of data increases the pressure on the hardware resources of the Grid as the tasks must have access to a larger amount of data. Prior to these updates, all jobs used a single CPU core and were single process. They were expected to match 2 GB of RAM, while their virtual memory could potentially reach up to 8 GB when swap space is included. In the new

*e-mail: marta.bertran.ferrer@cern.ch
**e-mail: costin.grigoras@cern.ch
***e-mail: rosa.m.badia@bsc.es

framework, given the high memory requirements, jobs can be multicore. Each such job can use up to the sum of the 2 GB RAM allocations per CPU allocated core. Multicore jobs are able to deploy multiple processes and threads that share memory resources and therefore use them more efficiently.

When the jobs managed through the new framework started to be executed on the Grid resources, it was detected that the efficiencies reported by the job monitor agents and by the machines in which they were executed differed widely. The ALICE Grid monitoring service is based on the MonALISA framework [1], which is composed of dynamic services that are used together to provide system monitoring, tracking of applications, tasks and services, serving also as a maintenance and optimisation tool for workflows and system control utilities. The sending of customised monitoring information to the MonALISA system is configured from the ApMon API and the UDP protocol is used to send this information.

For these initial jobs, some issues were detected in the CPU efficiency monitoring, as reported by MonALISA [1]. The observation of such anomalous behaviour triggered an analytical study of the workflows for an in-depth understanding of the root cause of the problem. Linux monitoring and tracing tools were used to analyse the internal job behaviour: the number and duration of processes and threads created, the use of the execution environment system resources and the overheads incurred. From this study the creation of a high amount of short-lived processes was spotted, some of which were too short-lived to be detected by the job monitoring methodology that was successfully being used in the legacy software stack.

This study led to the detection of areas with great potential for improvement in terms of more efficient resource utilisation and reduced overhead. Following a more in-depth look into the areas with the greatest potential, it was decided to approach the study starting with a general tracing of the workflow behaviour.

This paper is structured as follows: Section 2 gives an overview of commonly used workflow profiling and tracing methodologies. Section 3 presents the payload black-box analysis methodology with a focus on the payload process deployment, its nature, its temporal behaviour and its quantification. Section 4 describes the identified job execution potential optimisation areas, followed by observations resulting from correlation studies of the most relevant system parameters. Section 5 describes how the optimisations have materialised, the resulting improvements in the job deployment profile and the more efficient resource usage.

2 State of the Art

2.1 Profiling methodology and tools

A detailed analysis of system performance and running applications can lead to observations that serve as a basis for optimisation of code and resource consumption, increasing the scalability of the system and detecting problems such as bottlenecks or high latency levels [2]. A complete system analysis is based on the study of its different components, and it can use a wide range of tools that focus on monitoring specific components in the different layers of an application.

The tracing of running applications can be performed using static and dynamic tracing methodologies [3]. Static methodologies are based on the assignment of static trace statements to the entrance and exit points of functions. They are characterised by being a rigid methodology since the tracing points, once defined, cannot be deactivated during execution in cases where monitoring is not necessary. Moreover, static tracing points bring with them

a high overhead. Dynamic tracing methodologies, on the other hand, are capable of activating and deactivating the tracing of applications during their execution. In this category we find tracers of different nature according to their levels of intrusiveness, overhead and traced system levels.

The most widely used dynamic tracing tool to trace the execution of applications is *strace* [4]. This tool monitors the system calls executed and the signals received. From the generated logs, which contain the system call identifiers, their arguments and their return values, the behaviour of the running application can be analysed. It does not need a previous compilation of the code under study, thus having a great potential for the examination of user and kernel space boundaries, the detection of bugs, the identification of race conditions and sanity checking. As it does not perform regular function call tracing in the application nor record the calls made in the kernel mode, the level of detail of the logs is not very high. One of the main drawbacks of its use is the high overhead it adds to the execution, mainly due to heavy trap mechanisms and scheduling costs. The events that are collected in the trace must be preceded by an analysis from which meaningful observations are made, patterns are extracted and anomalous behaviour is detected. Depending on the level of complexity of the traces obtained, additional tools must be employed for a graphical visualisation from which relevant insights can be obtained.

Another widely used tool is *perf* [5]. Making use of hardware and software performance counters, *perf* provides users with the ability to monitor and study different aspects of application performance using a wide set of specific subcommands. It supports both dynamic and static tracepointing.

3 Observations and assessment of areas with optimisation potential

The ALICE Run 3 jobs are implemented on top of the O2 framework, which uses a distributed, parallel and staged data model [6]. This framework encompasses all the functionalities involved in high-energy physics experiments, such as calibration and readout, data recording and reconstruction, physics simulation and analysis.

3.1 Process deployment general tracing

The introduced CPU accounting issues were mainly observed in physics simulation workflows. These jobs are composed of chains of physics simulation - digitisation and reconstruction, which increases the process multiplicity with regards to other kind of productions. Therefore, they were not observed in analysis and reconstruction jobs, even though these also made the transition from single to multicore. This is due to the fact that their process deployment frequency was not as high, and they had longer lifetimes. Various tracing and profiling tools have been used to understand the workflow behaviour and to detect those areas with the greatest potential for optimisation.

The first level of study has been the tracing of deployed processes and threads during execution. The Linux tool *strace* [4] has been used to wrap the execution of the job, recording its logs to a file for later analysis and evaluation. This tracing cannot be performed routinely on all executed jobs on the Grid as one of the main drawbacks of *strace* is the high overhead involved, causing the job execution time to be extended by a factor of three.

At the end of the job execution, a Python script is used to change the format of the generated logs to *csv* for a more efficient data processing. From the analysis of the *csv* file, key metrics

have been obtained and plots have been generated to visualize the behaviour of the workflow and allow the extraction of meaningful insights and behavioural patterns.

The analysis presented in this paper focuses on the deep study of an 8-core simulation job implemented on top of the O2 simulation framework. An overview of its initial implementation is presented, as well as an assessment of areas with optimisation potential for a more efficient utilisation of the Grid computing resources.

3.1.1 Temporal study of job workflow

The analysis of the temporal behaviour of the execution has been performed by generating the Gantt plot of the deployed processes and threads, shown in Figure 1. This plot shows the dynamics of the creation and termination of the processes over time. One of the key observations has been the high number of processes that are executed in parallel. The fact that the executing machine does not have enough CPU cores to be running all of them in an uninterrupted way implies that part of them are in a *Sleeping* state. It also implies that the CPU power is distributed among these processes with continuous changes to the *Running* state, waking up the waiting processes. In total, more than 70,000 processes are spawned, most of which are very short-lived.

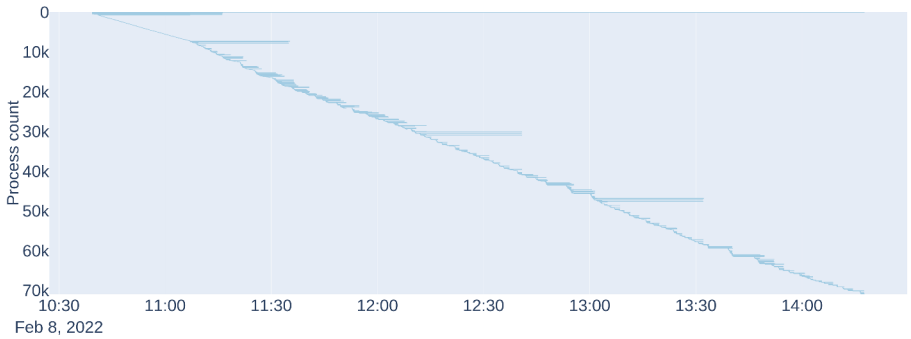

Figure 1: Gantt diagram of processes running in a simulation job.

3.1.2 Quantifying and identifying deployed processes

For an understanding of the types of processes that have been executed, they have been divided into categories according to their nature. Three categories have been identified: system commands, further classifying them by their system calls; O2 processes and spawned threads, categorised according to their parent process. Figure 2 shows in blue the deployed processes in each of the categories and in pink the corresponding number of deployed threads.

To relate the temporal study of the workflow to the number of deployed processes, the frequency with which they are spawned has also been studied, considering the execution time of the job without the added overhead of *strace*. The repetition of a process execution with a high frequency implies a large overhead derived from its initialisation time, as described in the following sections.

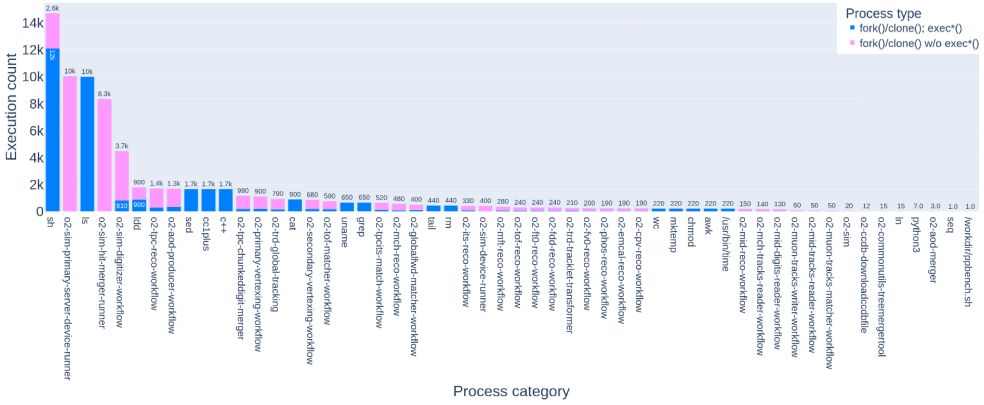

Figure 2: Process count sum per category with processes and threads without *exec*()* shown in parent process category.

4 Job execution limitations

Given the large number of multicore tasks that are expected to be executed on the ALICE Grid resources, software optimisations in such tasks are a key element to achieve a faster turnaround of the workflows concerned. The main sources of overhead are listed and discussed in the following subsections.

4.1 Frequent short process deployment

The ALICE Grid uses the MonALISA framework for monitoring and accounting of the used resources [1]. Before the LHC Run 3 jobs were single-process only, the accounting of CPU consumption and efficiency was adapted to those requirements. Having a single process executing ensured that the parsing of the output of the *ps* command was correct and reliable. With the change of workflow characteristics this methodology is no longer valid, because for the accounting of the totality of executed processes the sampling frequency would need to become significantly higher, bringing with it a considerable overhead.

The Linux */proc* file system keeps track of information about the running Linux system, both the kernel and the processes running [7]. The system keeps track of the cumulative user and system CPU time of the processes and all their children in the */proc/[PID]/stat* files. Whenever one of the child processes dies, its execution time is accounted for by its parent process. The newly implemented accounting methodology periodically analyses those contents for the existing processes to perform the global and per-process calculation of CPU time by summing their *usr* and *sys* CPU time components. The CPU efficiency is calculated by dividing the resulting value by the actual execution time of the job.

4.2 Large system CPU component consumption

The *system* component of the CPU usage accounts for the CPU consumed in low level tasks, such as interaction with hardware, communication between operating system processes, memory allocation, scheduling and interaction with the file system. In a highly efficient workflow, the consumption of *system* CPU should be low. In the workflow under analysis,

peak consumption of up to 25% of the CPU cores requested by the job is observed. The prolonged consumption of such high levels has large effects on the overall computational efficiency of the execution.

This high CPU consumption correlates with the deployment of the many short-lived processes which bring with them a large initialisation overhead. The origin of the many system calls is not identifiable at a glance, as the software stack on which these jobs are executed is made up of many dependent software layers. This makes it difficult to keep track of the followed workflow. The *gdb* debugging tool has been used for accurate tracking of the workflow of the target processes [8]. The processes that spawn system calls more frequently have been identified and their execution within the payload has been wrapped with a *gdb* script, while the target executables were replaced with symbolic links to such scripts. This methodology has been used to identify the origins of the most popular system calls.

The overhead of the initialisations of the many system calls in the job execution environment has been quantified. A considerable increase in initialisation time has been detected in this environment compared to their initialisation in the vanilla system. In terms of system time, this increase is quantified at 911%.

4.3 Heavy overhead in library loading

The overhead that comes with running in the job execution environment is mostly due to many failed attempts of *open* and *read* calls to load dependent libraries. This is caused by the incorrect configuration of the binary *rpath* and the environment variables *PATH* and *LD_LIBRARY_PATH*, all of which point to the paths which contain the executables and libraries. The exploration of a larger number of paths to locate the element to load implies a higher overhead.

Different strategies to reduce this overhead have been studied and evaluated by assessing the number of *open* system calls in the tested configurations. A significant reduction of these calls has been observed by creating a directory with the symbolic links to the libraries listed in the environment variable *LD_LIBRARY_PATH*, prepending it to the current list of directories. With this observation it is noted that changing the binary *rpath* to the final CVMFS path would increase the execution efficiency to values close to those for a local compilation. Based on these results, it is proposed to create a subdirectory in the working directory of the job where symbolic links are defined pointing to the executables and libraries resolved through the problematic environment variables. Adding symbolic links to the local directories */lib64* and */usr/lib* would also bring an improvement. The environment variables would have to be modified to take into account the new directories containing the symbolic links.

4.4 Concurrent processes

The amount of deployed child processes is high and there are periods during the execution when a large number of concurrent threads are running in parallel. The distribution of the allocated CPU power between the execution of these multiple processes leads to a large portion of them remaining in the *Sleeping* state and being woken up frequently to make use of the CPU for a given slot. These state changes involve a large number of context switches between them. Since these processes use shared memory, we can deduce that these context switches also involve a cost of serialisation and deserialisation in the accesses to the shared memory objects.

We have studied the ratio of voluntary and involuntary context switches and the CPU efficiency of the processes. It is concluded that in order to maximise CPU efficiency, a low ratio of voluntary versus non-voluntary context switches should be encouraged.

4.5 Incorrect efficiency accounting

Grid sites perform an independent accounting of the resources used by the executed workflows. The methodology employed to account for the CPU resources utilised by a job makes use of the output of the *time* command, which reports the amount of time spent executing in user and kernel space. From these values the CPU usage efficiency can be calculated as the ratio of the sum of the *usr* and *sys* components divided by the execution walltime. For this methodology to be valid, all child processes must remain connected to their parent in order to report their execution times in each of the components (*real*, *usr* and *sys*) as they finish. When a process dies, all its existing child processes get detached, and the parent of the defunct process will be unable to collect their execution times, which means they will be ignored and the resource accounting will be wrong.

Some situations have been detected in the execution pipelines where a process detachment was taking place, thus impacting the computation of the CPU efficiency due to the reporting of an extremely low CPU time.

4.6 Correlation of system parameters

In the analysed job, two of the system components with the highest potential to be optimised are the CPU *system* component consumed and the number of context switches. A Pearson correlation study [9] has been carried out on these two components with the rest of the monitored parameters, analysing the effects of their variations. A temporal correlation has been studied, resulting in clear correlations and anti-correlations of the components. Some of the main conclusions that have been drawn are the following:

- The amount of context switches is correlated with the fork creation rate and with the *system* CPU consumption.
- The context switch rate and the CPU *user* component are inversely correlated. Context switches also anti-correlate with the number of threads in the *RUNNING* state.
- The *system* CPU is highly correlated with the fork creation rate and the amount of disk read and write operations.
- The *system* and *user* CPU components maintain an inverse correlation. This is also the case with the number of processes and threads in the *RUNNING* state.

5 Improvements on payload workflows

Based on the observations from the analysis described in this paper, optimisations have been made to the payloads. These optimisations include a reduction of object initialisations, the merging of O2 executables, improved settings of the environment variables *LD_LIBRARY_PATH* and *rpath* of executables and the prevention of child process detachment from their parents.

The improvements observed in the execution of payloads have been remarkable, mainly in three areas:

- Reduction by 47% in the number of deployed processes and threads during job execution, from the initial 72.5K to 38.3K.
- Within this process count reduction, the pronounced decrease in the amount of system calls by 86% can be highlighted, followed by a 23% decrease in O2 processes and a 16% decrease in the total number of threads.
- A reduction of the job execution time by 35%, from 106 minutes to 69 minutes.

In addition to the improvements of the job payloads themselves, the monitoring of Grid job execution is now more complete and robust. Some of the parameters that were started to be monitored for the purposes of this study have been included in the regular monitoring workflow of the jobs, such as Context Switches or CPU instantaneous utilisation efficiency.

6 Conclusion

This paper presents the updated CPU usage accounting methodology of the ALICE Grid to efficiently account for all the resources used in different task execution scenarios. The development of the new methodology has led to the improvement of the real-time detailed monitoring of jobs. Observations made from the extended monitoring have been used as a source to identify areas of potential payload optimisation.

In addition to the parameters extracted from the resource monitoring of jobs, an in-depth analysis of the internal mechanics of deployment and execution of processes has been undertaken. As a result of this introspection into the payload workflow, significant improvements have been introduced in several areas of the framework. Firstly, a better understanding of the code behaviour and process count has been achieved, which has led to an improvement in the CPU utilisation of Grid jobs and a considerable reduction in their execution times.

Undertaking such an immersive study into the internal workflows of the Grid tasks has proven to be a powerful tool for spotting optimization areas. Our goal is to continue studying payloads and introducing new optimisations using the profiling methodology implemented, starting from a detailed monitoring of the jobs as a source for uncovering potential areas of optimisation that encourage the deployment of improvements in the physics software framework.

7 Bibliography

References

[1] C. Grigoras, R. Voicu, N. Tapus, I. Legrand, F. Carminati, L. Betev, *MonALISA-based Grid monitoring and control* (Springer, 2011), Vol. 126, pp. 1–7

[2] B. Gregg, *Linux performance analysis and tools* (2013)

[3] A.V. Mirgorodskiy, B.P. Miller, *Autonomous analysis of interactive systems with self-propelled instrumentation*, in *Multimedia Computing and Networking 2005*, edited by S. Chandra, N. Venkatasubramanian, International Society for Optics and Photonics (SPIE, 2005), Vol. 5680, pp. 188 – 202

[4] *strace(1) — Linux manual page*, https://man7.org/linux/man-pages/man1/strace.1.html, [Online; accessed 17-May-2022]

[5] A.C. De Melo, *The new linux'perf'tools*, in *Slides from Linux Kongress* (2010), Vol. 18, pp. 1–42

[6] P. Buncic, M. Krzewicki, P. Vande Vyvre, *Technical Design Report for the Upgrade of the Online-Offline Computing System. CERN-LHCC-2015-006, ALICE-TDR-019* (2015)

[7] *proc(5) — Linux manual page*, https://man7.org/linux/man-pages/man5/proc.5.html, [Online; accessed 21-October-2022]
[8] R. Stallman, R. Pesch, S. Shebs et al., *Debugging with GDB* (1988), Vol. 675
[9] J. Benesty, J. Chen, Y. Huang, I. Cohen, *Pearson correlation coefficient*, in *Noise reduction in speech processing* (Springer, 2009), pp. 1–4