EPJ Web of Conferences **214**, 03006 (2019) https://doi.org/10.1051/epjconf/201921403006
CHEP 2018

Improving efficiency of analysis jobs in CMS

Todor Trendafilov Ivanov[1,*], *Stefano* Belforte[2], *Matthias* Wolf[3], *Marco* Mascheroni[4], *Antonio* Pérez-Calero Yzquierdo[5,6], *James* Letts[4], *José M.* Hernández[5], *Leonardo* Cristella[7], *Diego* Ciangottini[8], *Justas* Balcas[9], *Anna Elizabeth* Woodard[3], *Kenyi* Hurtado Anampa[3], *Brian Paul* Bockelman[10], and *Diego* Davila Foyo[11] for the CMS Collaboration

[1]University of Sofia, Sofia, Bulgaria
[2]Università e INFN Trieste, Trieste, Italy
[3]University of Notre Dame, Notre Dame, IN. USA
[4]University of California San Diego, La Jolla, CA, USA
[5]Centro de Investigaciones Energéticas Medioambientales y Tecnológicas (CIEMAT), Madrid, Spain
[6]Port d'Informació Científica (PIC), Barcelona, Spain
[7]INFN Bari, Bari, Italy
[8]Università e INFN Perugia, Perugia, Italy
[9]California Institute of Technology, California, USA
[10]University of Nebraska-Lincoln, Lincoln, NE, USA
[11]Benémerita Universidad Autónoma de Puebla, Puebla, México

Abstract. Hundreds of physicists analyze data collected by the Compact Muon Solenoid (CMS) experiment at the Large Hadron Collider using the CMS Remote Analysis Builder and the CMS global pool to exploit the resources of the Worldwide LHC Computing Grid. Efficient use of such an extensive and expensive resource is crucial. At the same time, the CMS collaboration is committed to minimizing time to insight for every scientist, by pushing for fewer possible access restrictions to the full data sample and supports the free choice of applications to run on the computing resources. Supporting such variety of workflows while preserving efficient resource usage poses special challenges. In this paper we report on three complementary approaches adopted in CMS to improve the scheduling efficiency of user analysis jobs: automatic job splitting, automated run time estimates and automated site selection for jobs.

1 Introduction

The Compact Muon Solenoid (CMS) experiment [1] at CERN requires extensive capabilities for data processing, Monte Carlo simulation production, and user analysis tasks. The CMS Remote Analysis Builder (CRAB) is the user analysis workflow management tool. The initial focus of the CRAB project was to provide access to the resources of the Worldwide LHC Computing Grid (WLCG) [2] and to provide the capability to perform analytical tasks to several hundred unique users per month. A performance of 40,000 simultaneously running jobs and a daily completion rate of 500,000 analysis jobs have been achieved (Fig. 1). This contribution describes the current efforts focused on improving CPU utilization and scalability, and reducing user workflow turnaround time. These optimizations must be achieved without requiring any modifications of user applications.

*Corresponding author, e-mail: todor.trendafilov.ivanov@cern.ch

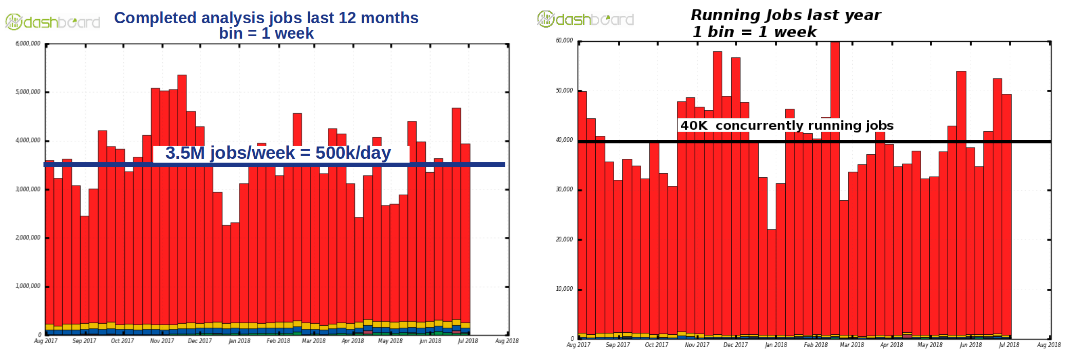

Figure 1: Number of completed analysis jobs (in red) per week (left) and of concurrently running analysis jobs (right) for a 12 months period. Other colors indicate test jobs.

2 CMS Computing on the Grid

2.1 Compute resource provisioning

The workload management system of the CMS experiment executes payloads in compute nodes provisioned through GlideinWMS [3] and thus made available as execution slots in a Vanilla Universe HTCondor [4] pool which we refer to as the Global Pool [5, 6]. From the infrastructural point of view, a workflow goes through a few sequential steps:

- Users submit HTCondor jobs via specific workload management tools: WMAgent for central data processing and Monte Carlo production jobs, and CRAB for user jobs.
- The GlideinWMS monitors idle jobs in the global pool and sends pilot jobs, called *glideins*, to the Grid sites as needed. Pilots usually run for 48 hours on 8 core. Each pilot runs an HTCondor *startd* which joins the CMS *Global Resource Pool* for the time the pilot job has to live and pulls jobs from the global queue as needed dynamically allocating its 8 cores to several multi/single-core jobs and reallocating freed up cores until the end of its lifetime.
- The users jobs get matched to the free slots announced by the pilots.

This architecture provides a well structured resource pool with explicitly defined building blocks, and gives a relatively homogeneous view on the otherwise extremely heterogeneous grid resources. In this picture, CMS takes ownership for all issues of the resource pool itself, like fragmentation due to running variable number of multi/single-core jobs of different length, waste of CPU time because of lack of proper pilot jobs utilization, etc. Such schema provides hooks for optimization at the cost of making inefficiency more visible. More details in: [5, 6].

2.2 Analysis jobs management: CRAB

CRAB is the software system that interfaces directly with CMS users and accepts analysis requests that specify an executable analysis program and a set of data files (called a *dataset*) to which it will be applied. This is referred to, in CRAB, as a *task*.

A dedicated machine running a component of CRAB called TaskServer divides the user request into segments called jobs, each of which specifies the code and data required to execute a portion of the task on a single worker node. This process, by which a single user request is segmented into a set of potentially thousands of jobs is called *splitting*. Users can specify one among a few different algorithms used for the splitting, for example 'EventBased' or 'FileBased'. TaskServer prepares a specification for each task as a Directed Acyclic Graph (DAG). The DAG is then submitted to one of a set of HTCondor schedulers where it is executed under control by the HTCondor DAGMAN, a high-level scheduler which will submit grid jobs to the schedd as needed. When all jobs in a single task complete, the Asynchronous Stage Out (ASO) component moves the job outputs from their remote storage locations to the user's preferred final location. Metadata describing the task output is stored in the CMS Dataset Bookkeeping System [7], allowing other researchers to access it. An overview of the CRAB architecture is provided in Fig. 2.

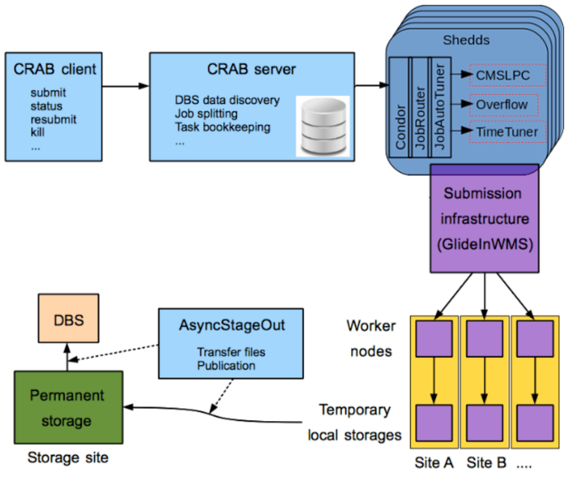

Figure 2: Overview of the CRAB architecture.

3 Scheduling optimization

Performance has been improved through three lines of development:

- **Automatic Splitting:** Jobs which are excessively long result in inefficient distribution of work, delaying completion of the task, or waste of resources, when they get killed by an expiring pilot or a random glitch in the infrastructure. Jobs which are excessively short cause unnecessary loads on the infrastructure. CRAB has been revised to optimize the task splitting process so that most jobs are close to the optimal length target, which is set at eight hours based on experience gained while driving the infrastructure.
- **Time Tuning:** Each pilot job slot allocation is 48 hours in length, and is assigned several payload jobs, leaving a variable amount of available processing time at the end of the slot. Time tuning minimizes any time left unused by assigning a final job that fits within the available time.
- **Overflow:** Jobs are normally assigned to sites that host the input data each job requires, however jobs can also read data from a remote site via the XrootD [8] service. Work queued for execution at busy sites can be overflowed, or transferred to resources elsewhere on the Grid that are less busy, and then access data as needed across the wide area network.

3.1 Automatic Splitting

3.1.1 Theory

Before this optimization was introduced, each task was processed according to a single static DAG specifying a fixed set of jobs. The splitting parameters configured manually by the users often resulted in thousands of very short jobs which would run for only a few minutes (see Fig. 3), causing an excessive infrastructure load, or in a few very long jobs delaying the completion of the task. With the automatic task splitting a series of DAGs is generated with different granularity at each step, allowing the task to be segmented more efficiently into a smaller number of jobs of more uniform length:

- A *probe DAG* runs a few (typically 5) short jobs to estimate time, memory, and disk storage requirements. Splitting parameters are computed based on the processing time per event determined from the probe output. The nominal target runtime is 8h per job.
- A *processing DAG* is created to run the jobs produced by splitting the task according to the computed parameters. These jobs are assigned a limited runtime; jobs which are not completed will return the detail of what is still to be analyzed.
- Up to 3 *tail DAGs* (one when 50% of the processing DAG is done, one at 80% and one up to the end) re-split and execute failed jobs and remaining work from the processing step.

3.1.2 Practice

The automatic splitting approach has required significant operational changes. The previous user-driven splitting process runs on a single central server, before jobs are instantiated and submitted, therefore it is easy to access logs and to replay actions for troubleshooting and debugging. In the new model, splitting is done separately for each task in the fifteen HT-Condor schedulers that are currently accessed by CRAB. Changes continue to occur over the lifetime of each task and are dependent on real-time changes in performance across the grid, making problems difficult to reproduce. Consequently we have adopted an incremental deployment process to allow users to become familiar with each feature and allow us to identify and correct any problems before moving on to the next modification.

The current implementation has been in production since February 2018. Users are encouraged but not pushed to use it and current adoption is about 2%. So far only a few minor issues have been faced. Extending usage requires an education campaign, i.e. substantial manpower. Fig. 4 shows a clear peak at a job runtime of eight hours, which corresponds to the default value for the automatic splitting mechanism.

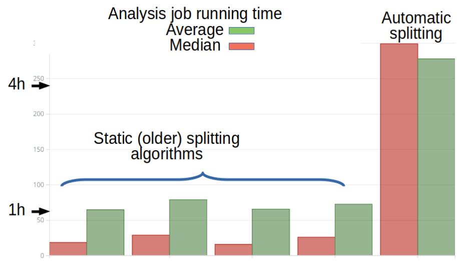

Figure 3: Analysis jobs median run time.

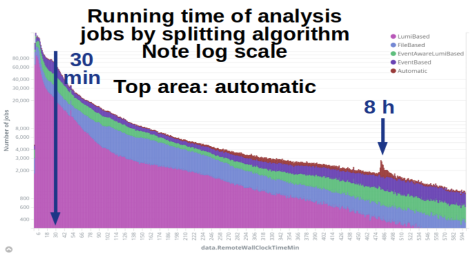

Figure 4: Analysis jobs run time distribution.

3.2 Automatic Tuning

Both *Time Tuning* and *Overflow* have been implemented using the same underlying mechanism called Automatic Tuning, where an automated process periodically monitors the requirements of jobs idle in the HTCondor queue and modifies them according to predefined rules. This makes it possible to optimize the initial requirements in light of actual job performance and sites availability. Job requirements are modified with the HTCondor JobRouter, which scales much better than a massive *condor_qedit*. It is frequently necessary to choose between collecting additional global information so that the tuning process can better optimize the analysis, and making changes in the HTCondor schedulers based on the information already collected. It is also important to minimize work done on the scheduler to avoid interfering with the job matching and starting process.

Our approach was to use a central process to collect information and generate statistics based on a feed of HTCondor classAds to Elastic Search, where the global view is updated every 12 minutes. A lightweight script then runs on each scheduler and uses the HTCondor *JobRouter* to do the actual classAd remapping locally.

This strategy has been implemented for central data processing and simulation tasks, in which the workflows are larger and there is more *top-down control*. In the user analysis domain, although most workflows are smaller, the number of workflows is much larger and more diverse, making it more difficult to identify and correct inefficiencies. The process of gathering performance data is slow, and changes in system parameters take time to propagate, and recovery from configuration errors is difficult. These factors lead us to make changes in a measured and incremental fashion.

3.2.1 Time Tuning: Theory

Unless automatic splitting was used, jobs are initially submitted to HTCondor with a very rough estimate of the needed walltime in the *MaxWallTime (MWT)* classAd attribute specified for all jobs in a task. HTCondor teminates any job that reaches this value. It is set at the time of submission and persists until the job is either completed or terminated. The median job run time is about 30 minutes, however most jobs are submitted with the default MWT value of 20 hours since run time can vary considerably among the jobs in a task as a result of variations in computational requirements, assigned hardware, and I/O latency, and users must submit a relatively high MWT to allow all jobs to complete. This is problematic because HTCondor will not schedule jobs in pilot slots which do not have the full MWT remaining for available execution time. It is therefore likely that multicore pilot slots remain unused, resulting in wasted resource usage.

Consequently we have introduced a new classAd attribute called EstimatedWallTime (EWT). EWT is calculated to be as close as possible to the real job Run Time (RT) and is used to schedule jobs, while the originally defined MWT is used to kill jobs and purge them from the queue. Jobs can be scheduled in any slot which has at least the EWT time remaining (see Fig. 5 - left). MWT is still set as before, but a job can be initiated even if the MWT exceeds the available time in the slot. If EWT is exceeded but time remains in the pilot slot, as is usually the case, the job continues to run. If a job exceeds either the available time in the slot (see Fig. 5 - right) or the MWT it will be terminated and will be automatically resubmitted as explained below, however with accurate EWT this affects only a small fraction of the jobs.

3.2.2 Time Tuning: Practice

An initial value of EWT for a task is computed as soon as at least one job is completed, and EWT is then dynamically updated every 10 minutes. It is set at the 95th percentile of the distribution of running times for the jobs in the task that have been completed. This statistical approach has limitations: while some CRAB tasks result in thousands of jobs, most have a few hundred or fewer jobs, so the estimate of running time may be imprecise. The estimate may also be biased because the first jobs which complete in a task and are used for the initial run time estimate are likely to be those which need less data or are assigned to particularly efficient nodes. To minimize this error the estimate derived from the distribution of completed jobs is increased by a correction factor dependent on the number of jobs. The EWT classAd attribute is then added to each job and periodically updated by the JobRouter. Typically, EWT develops into one hour or less while pilot jobs run for 48 hours. If a job is still running when the pilot shuts down when reaching the end of its lifetime, the job is terminated and automatically rescheduled by HTCondor. CPU time used on a job which is terminated is wasted. Our task is to minimize the combination of CPU time wasted on terminated jobs and the CPU wasted on unused pilot slots, see: Fig. 5.

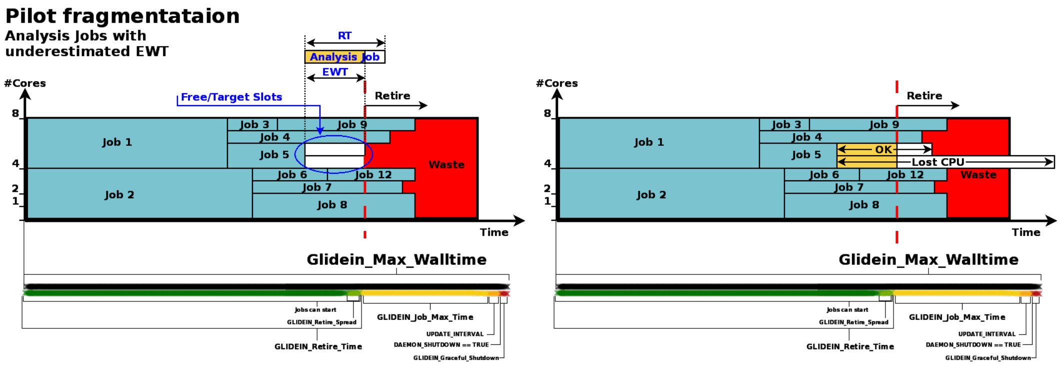

Figure 5: Multicore pilots *Time fragmentation* - free slots occupied by analysis jobs. Here *Retire* indicates the period of the pilot's lifetime during which no new jobs are accepted.

Automated tuning of the job time request process was rolled out in production in Spring 2018. To minimize the possibility of interference with ongoing analysis of LHC Run2 data, we used a conservative correction factor for EWT under which 95% of jobs still ran in less than EWT. An additional 4% of jobs exceeded the EWT but completed before the pilot expired. Only 1% of all jobs reached the pilot end of life and were restarted by HTCondor; we found that all such jobs completed after a single restart. Figure 6 illustrates how this substantially improved utilization of the final portion of the pilot slot. Jobs which were not time-tuned could only be scheduled for pilot slots which had more than 21 hours of time remaining, while jobs whose time requirement had been tuned by the JobRouter were evenly distributed across the duration of each slot.

3.2.3 Overflow: Theory

Network latency is lower and LAN bandwidth higher between storage and computing resources at a common site, so jobs are preferentially submitted to sites hosting the required

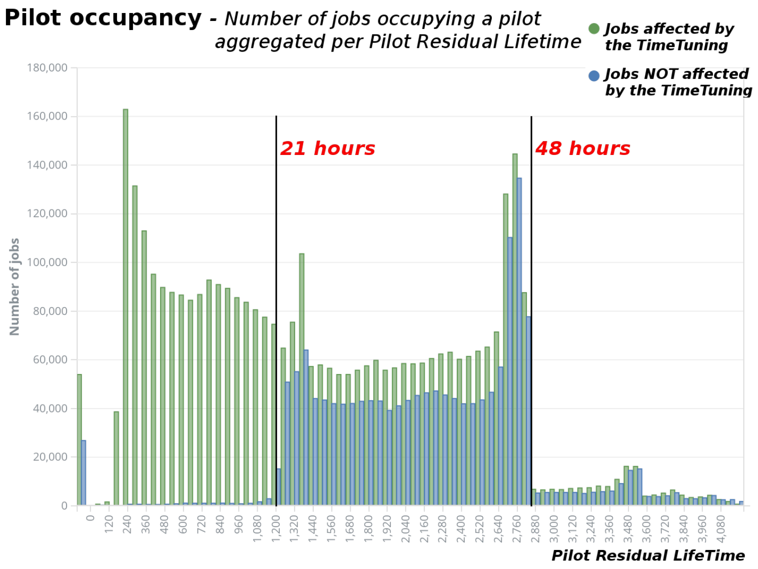

Figure 6: Pilot execution slot occupancy vs. remaining pilot life time. The bins above 48h represent the small fraction of pilots which run at sites allowing long running times.

input data in local storage. However, sites hosting popular data receive a much larger number of job requests than the available CPU slots, leading to long job waiting times.

The CMS global XrootD data federation [8] allows jobs to access data remotely through the WAN, at the cost of a modest loss in CPU efficiency due to I/O delays. Properly scheduling such jobs while considering the limitations on network bandwidth between sites is a complex task. We have attempted a simple approach, based on the JobRouter, that can enable remotely-reading jobs by dynamically expanding the list of potential job execution sites (overflow) beyond the sites hosting the input data.

3.2.4 Overflow: Practice

The CMS computing model [9–11] follows a hierarchical structures where few larger sites (Tier-1) mostly handle organized data processing (production) while a constellation of smaller sites (Tier-2) is the favored place for user workflows (analysis). Since Tier-1 sites have large disk pools, but limit to 5% the amount of CPU resource available to analysis, there are frequent cases where input data for analysis is only hosted at a Tier-1 and user jobs are queued for a long time. We have addressed this by using overflow mechanism to allow of jobs initially scheduled to be executed at Tier-1 sites to run at Tier-2 sites located in the same country. This regional strategy makes additional computing resources available while minimizing the increased latency and network load that would result from international access. A typical situation with one overloaded Tier-1 site is illustrated in Fig. 7 which shows where jobs initially queued for a Tier-1 site were actually executed.

4 Conclusions and directions for further work

The goal of CRAB is to provide researchers in high energy physics with the ability to apply the distributed worldwide resources of the CERN computing grid to analysis tasks ranging

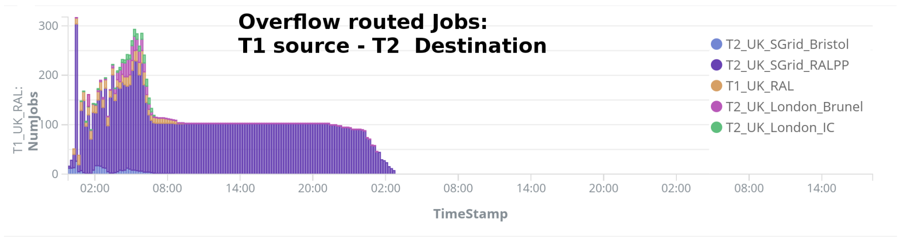

Figure 7: Overflow from the Tier-1 Site: “T1_UK_RAL” to all Tier-2 sites in the UK region.

from simple to massive in scope. This requires that complex tasks be split into individual jobs that can be distributed among dozens, hundreds, or thousands of machines around the world as they become available. Optimizing the performance of such a complex network of resources requires making inferences about the needs of computational tasks based on limited sampling of the code and data while simultaneously adapting to real-time changes in available resources and network conditions. Updates to the system must be introduced in an incremental manner to minimize disruption of ongoing research. Effective monitoring of many aspects of system performance is essential to identify and correct problems at an early stage.

Meeting the computational requirements of researchers in high energy physics is a daunting task. While CRAB has made significant advances, our goal is to continue to improve performance, efficiency, and user friendliness. This will require the application of new strategies including local control, network aware scheduling, and machine learning.

References

[1] The CMS Collaboration, “The CMS experiment at the CERN LHC”, *JINST* **3**, S08004 (2008), doi 10.1088/1748-0221/3/08/S08004

[2] I. Bird, “Computing for the Large Hadron Collider”, *Annual Review of Nuclear and Particle Science*, **61:99–118** (2011), doi 10.1146/annurev-nucl-102010-130059

[3] I. Sfiligoi et al. “The Pilot Way to Grid Resources Using glideinWMS”, *WRI World Congress on Computer Science and Information Engineering*, Vol. **2** 428-432 (2009)

[4] D. Thain, T. Tannenbaum, and M. Livny, “Distributed Computing in Practice: The Condor Experience“, *Concurrency and Computation: Practice and Experience*, Vol. **17**, No. 2-4, pages 323-356, February-April, (2005).

[5] J. Letts et al. “Improving the Scheduling Efficiency of a Global Multi-core HTCondor Pool in CMS”, **to be published in these proceedings**.

[6] A. Perez-Calero Yzquierdo et al. “Exploring GlideinWMS and HTCondor scalability frontiers for an expanding CMS Global Pool”, **to be published in these proceedings**.

[7] M. Mascheroni et al. "CMS distributed data analysis with CRAB3", *J. Phys.: Conf. Ser.* **664** 062038 (2015)

[8] J. Balcas et al. "HTTP as a Data Access Protocol: Trials with XrootD in CMS’s AAA Project", *J. Phys.: Conf. Ser.* **898** 062042 (2017)

[9] M. Aderholz et al. “Models of Networked Analysis at Regional Centres for LHC Experiments (MONARC) - Phase 2 Report”, CERN-LCB-2000-001 (2000).

[10] C. Grandi et al. “The CMS Computing Model”, CERN-LHCC-2004-035, CMS-NOTE-2004-031, LHCC-G-083.

[11] C. Grandi et al. “CMS computing model evolution”, *J. Phys.: Conf. Ser.* **513** 032039 (2014)