PROCEEDINGS OF SCIENCE

Improving efficiency of analysis jobs in CMS

Leonardo Cristella[*†]
Università e INFN Bari, Bari, Italy
E-mail: leonardo.cristella@cern.ch

Data collected by the Compact Muon Solenoid experiment at the Large Hadron Collider are continuously analyzed by hundreds of physicists thanks to the CMS Remote Analysis Builder and the CMS global pool, exploiting the resources of the Worldwide LHC Computing Grid. Making an efficient use of such an extensive and expensive system is crucial. Supporting a variety of workflows while preserving efficient resource usage poses special challenges, like: scheduling of jobs in a multicore/pilot model where several single core jobs with an undefined run time run inside pilot jobs with a fixed lifetime; avoiding that too many concurrent reads from same storage push jobs into I/O wait mode making CPU cycles go idle; monitoring user activity to detect low efficiency workflows and provide optimizations, etc. In this contribution we report on two novel complementary approaches adopted in CMS to improve the scheduling efficiency of user analysis jobs: automatic job splitting and automated run time estimates. They both aim at finding an optimal value for the scheduling run time. We also report on how we use the flexibility of the global CMS computing pool to select the amount, kind, and running locations of jobs exploiting remote access to the input data.

International Symposium on Grids & Clouds 2019, ISGC2019
31st March - 5th April, 2019
Academia Sinica, Taipei, Taiwan

[*]Speaker.
[†]on behalf of the CMS collaboration.

© Copyright owned by the author(s) under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0).

PoS(ISGC2019)015

1. Introduction

The analysis workflow of data collected by the Compact Muon Solenoid (CMS) experiment [1] at CERN involves computational and data intensive steps: from data processing, through Monte Carlo simulation production, to user analysis tasks. The latter is accomplished via an analysis workflow management tool: the CMS Remote Analysis Builder (CRAB). CRAB focuses on providing the capability to execute analysis tasks to several hundred unique users per month via the resources of the Worldwide LHC Computing Grid (WLCG). A scale 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 in CRAB to improve CPU utilization and scalability, and reduce user workflow turnaround times without requiring any modifications of user applications [2]. Strategies, concepts, details, and operational experiences are discussed, highlighting the pros and cons and showing how such efforts have helped to improve the computing efficiency in CMS.

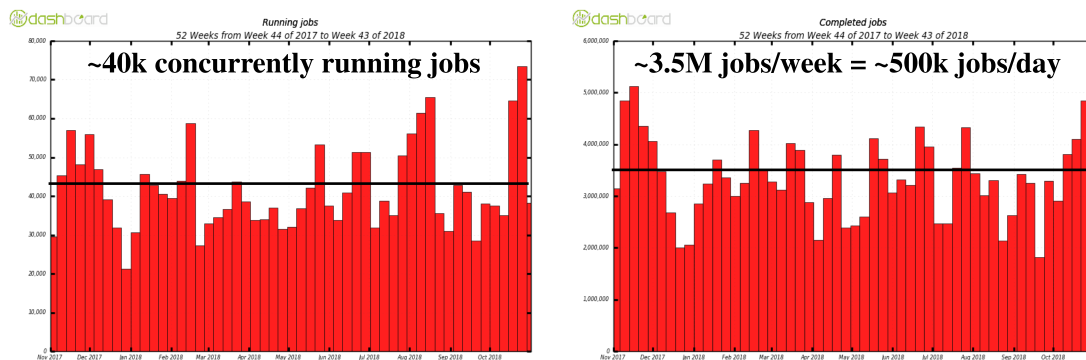


Figure 1: Number of concurrently running analysis jobs (left) and number of completed analysis jobs per week (right) for a 12 months period.

2. CMS Computing on the Grid

2.1 Compute resource provisioning

The Glidein workload management system (WMS) [3] acquires and provides the compute nodes on which CMS payloads are executed, made available as execution slots in a Vanilla Universe HTCondor [4] pool, referred to as the CMS Global Pool [5, 6]. From the infrastructural point of view, a data processing workflow goes through a few sequential steps:

- Two specific workload management tools - WMAgent for central data processing and Monte Carlo production jobs, and CRAB for user jobs - submit HTCondor payload jobs based on users requests.
- Based on idle jobs in the global pool, the GlideinWMS sends pilot jobs, called *glideins*, to the Grid sites as needed. An HTCondor *startd* which joins the CMS *Global Resource Pool* is run by each pilot and pulls jobs from the global queue, which contains both WMAgent and CRAB payloads. Usually a pilot acquires 8 cores for 48 hours which are dynamically

PoS(ISGC2019)015

allocated (and reallocated as freed up) to several multi/single-core jobs until the end of its lifetime via HTCondor match-making.

The WLCG resources are heterogeneous and in this provisioning model CMS takes ownership for all scheduling issues of the global pool, like fragmentation due to variable number of multi/single-core jobs of different length (WMAgent usually requiring more resources than CRAB), waste of CPU time because of lack of proper pilot jobs utilization, etc. The resource pool defined by such architecture is well structured with explicitly defined building blocks, providing hooks for optimization at the cost of making inefficiency more visible. More details can be found in [5] and [6]. For central data processing and simulation tasks, workflows are larger, well structured and with more *top-down control*, which makes them less exposed to inefficiency sources. In the user analysis domain instead, most workflows are smaller, the number of workflows is much larger and more diverse than in central production, making them more prone to inefficiencies.

2.2 Analysis jobs management: CRAB

CMS users often need to run an analysis application over a large set of data (called a *dataset*). CRAB is the software system that turns such request into a set of jobs executed on the grid, referred to as a *task*. CRAB prepares, submits, monitors and bookkeeps the task execution and transfers and register output files. Fig. 2 provides an overview of the CRAB architecture which is further explained in ref. [7]. The process by which CRAB breaks a monolithic user request into a set of potentially thousands of jobs is called *splitting* and happens on the *TaskServer* component of CRAB. Each job specifies the information required to execute a portion of the task on a single worker node. Users can specify one among a few available splitting algorithms, for example 'EventBased' or 'FileBased'. Then a Directed Acyclic Graph (DAG) is created by the TaskServer for each task and submitted to one of a set of HTCondor schedulers where it is executed under control by the HTCondor DAGMAN, a high-level scheduler with the role of submitting grid jobs to the worker nodes. Eventually, job output files are moved from the local storage of the job execution site to the user preferred final storage location by the Asynchronous Stage Out (ASO) service, recording files metadata in the CMS *Dataset Bookkeeping System* (DBS) which makes the task output usable as input for further processing.

3. Scheduling optimization

Three independent lines of development have been pursued in order to improve the resource usage efficiency:

- **Automatic Splitting:** Few hours is the optimal runtime for jobs in WLCG. Jobs which are excessively long can get killed by an expiring pilot or a random glitch in the infrastructure, resulting in an inefficient distribution of work, delayed completion of the task, or waste of resources. Jobs excessively short cause unnecessary loads on the infrastructure and suffer from startup overheads. The Automatic splitting does not require the user to specify any splitting parameter and optimizes jobs for optimal time duration.

PoS(ISGC2019)015

- **Time Tuning:** The 48 hours pilot slot is allocated with several payload jobs having different run times. The amount of the available processing time left unused is wasted, therefore, a good estimation of the job running time is essential for an efficient utilization of the slots. The dynamic tuning of this parameter based on finished jobs of the same task increases the utilization efficiency by optimizing the *jobs to pilot slot allocation*

- **Overflow:** Jobs are normally assigned to sites that host the input data that each job requires. Several factors such as data popularity, number of local users, etc., generate an uneven workload among the grid sites resulting in long job queues. Since data can be read also remotely via the XrootD [8] service, albeit with a somewhat smaller CPU efficiency, the overflow system optimizes the *jobs scheduling across sites* by transferring work queued for execution to nodes that are less busy elsewhere on the grid.

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 (see Fig. 3a) which would run for only a few minutes, 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:

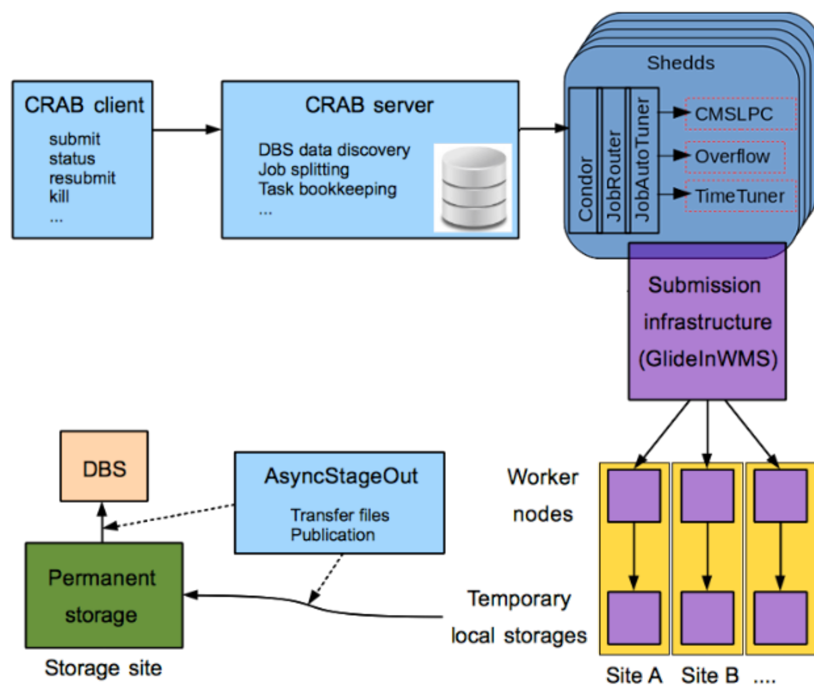
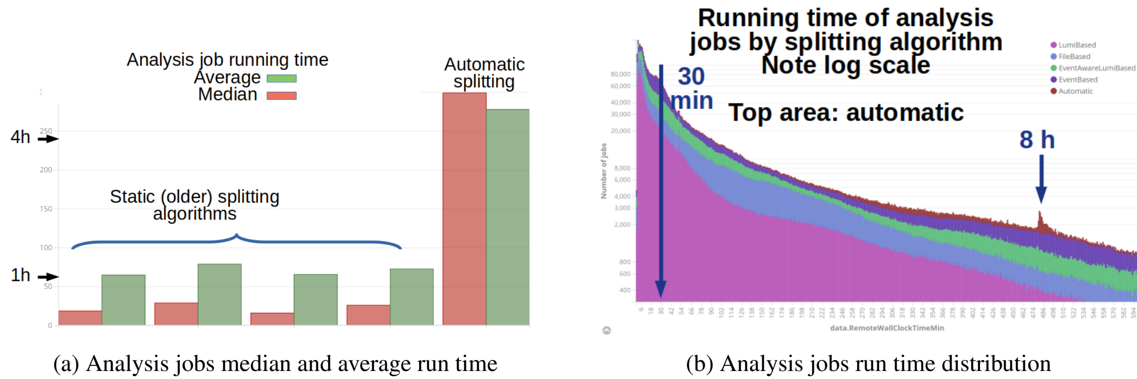


Figure 2: Overview of the CRAB architecture.

PoS(ISGC2019)015



(a) Analysis jobs median and average run time

(b) Analysis jobs run time distribution

Figure 3: Analysis jobs run time.

- A *probe DAG* runs a few short jobs (typically 5) to estimate time, memory and disk storage requirements. Some of these parameters depends on the worker node the jobs run on and represent the only machine benchmark in use. Splitting parameters are computed based on the processing time per event determined from the probe output. The nominal target run time 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 fixed maximum run time; jobs that run out of time before completing the assigned work are gracefully terminated. The remaining data will be processed in the next step.
- 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 was run on a single central server, before jobs were instantiated and submitted, therefore it was 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 HTCondor schedulers that are currently used 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 stage.

The current implementation has been in production since February 2018. Users are encouraged but not forced to use it and consequently the current adoption is low, about 2%. So far only a few minor issues have been faced. Extending usage requires an education campaign, hence substantial effort. Fig. 3b shows the analysis job run time distribution. A clear peak at eight hours is visible, which corresponds to the default value for the automatic splitting mechanism.

PoS(ISGC2019)015

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 to better fit available execution resources. This makes it possible to optimize the initial requirements in light of actual job performance and sites availability.

It is frequently necessary to choose between collecting additional global information, so that the tuning process can better optimize the job scheduling, 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 does the actual classAd remapping locally via the HTCondor JobRouter, which scales much better than a massive *condor_qedit*.

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, instead most workflows are smaller, the number of workflows is much larger and more diverse than in central production, making it more difficult to identify and correct inefficiencies. The process of gathering performance data can be slow compared to the running time of a given DAG, changes in system parameters take time to propagate and recovery from configuration errors is difficult. These factors led 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 terminates 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 measured median running time for analysis jobs 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, therefore users must set 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 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. MWT is still set as before, but a job can be initiated even if the MWT exceeds the available time in the slot. If the 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, it will be terminated and automatically resubmitted. We have achieved an accurate enough estimation of EWT such that this affects only a small fraction of the jobs.

PoS(ISGC2019)015

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. EWT is set at the 95th percentile of the distribution of running times for the jobs in a 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 to process 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 an empiric 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 stabilizes to 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. The time tuning algorithm tries to minimize the combination of CPU time wasted on terminated jobs and the CPU wasted on unused pilot slots (see Fig. 4).

Automated tuning of the requested job running time was rolled out in production in Spring 2018. To minimize the possibility of interference with ongoing analyses 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. Fig. 5 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 lifetime 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 bandwidth higher between storage and computing resources located at a common site. For this reason, jobs are preferentially submitted to sites hosting the

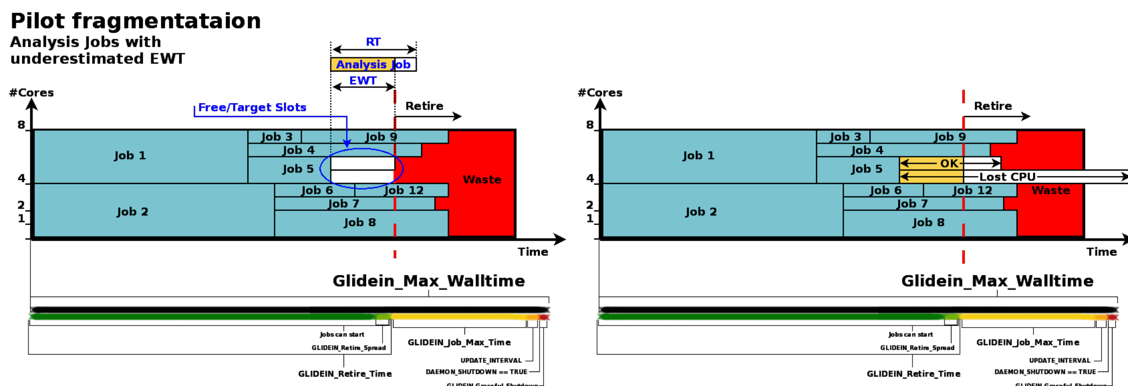


Figure 4: Multicore pilots *Time fragmentation* - free slots occupied by analysis jobs.

PoS(ISGC2019)015

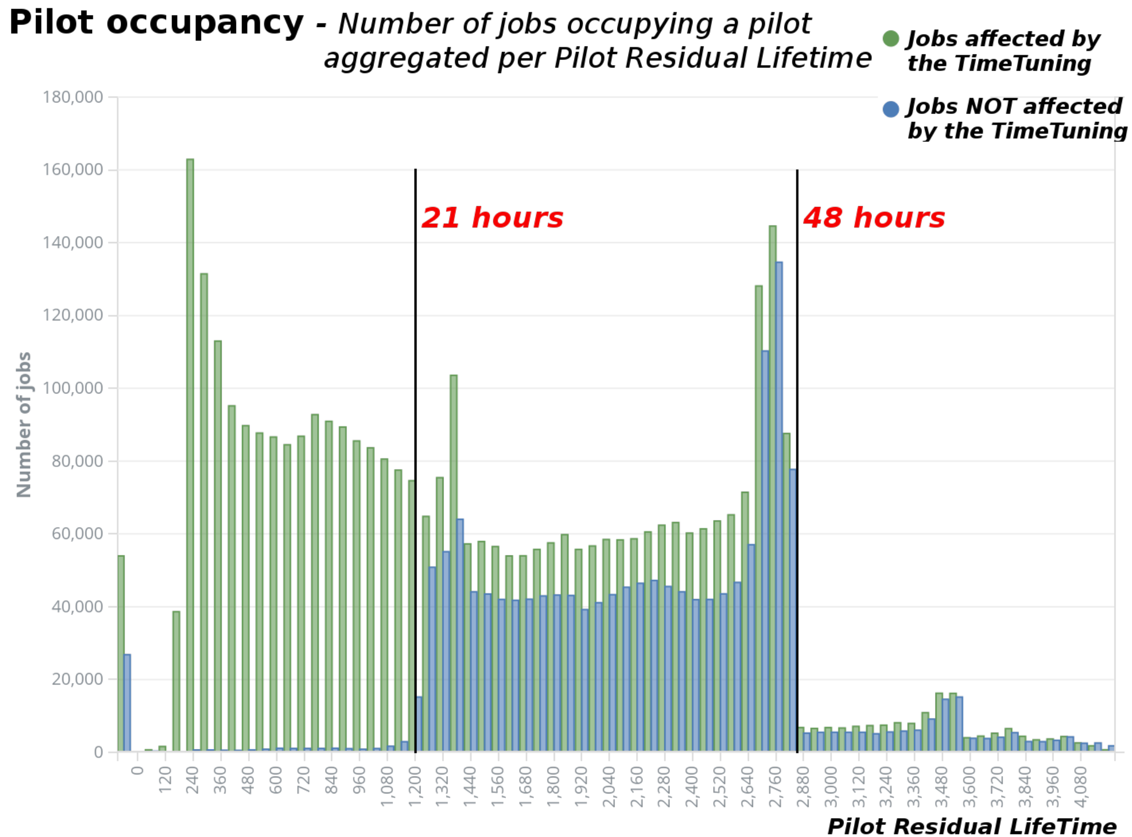


Figure 5: Pilot execution slot occupancy vs. remaining pilot life time. The bins above 48h represent the small fraction of pilots which run at sites allowing long running times

required input data in local storage. However, sites hosting popular datasets usually 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 increased I/O latency. Properly scheduling such jobs while considering the limitations on the WAN network 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

We have initially addressed the most pressing need for user analysis: Tier-1 sites often host interesting datasets, however the share of computing resources available for analysis jobs at such sites is only 5%. We have enabled 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 over a 10-days period is illustrated in Fig. 6. The top plot shows the number of idle analysis jobs at the French Tier-1 site, the bottom plot shows

PoS(ISGC2019)015

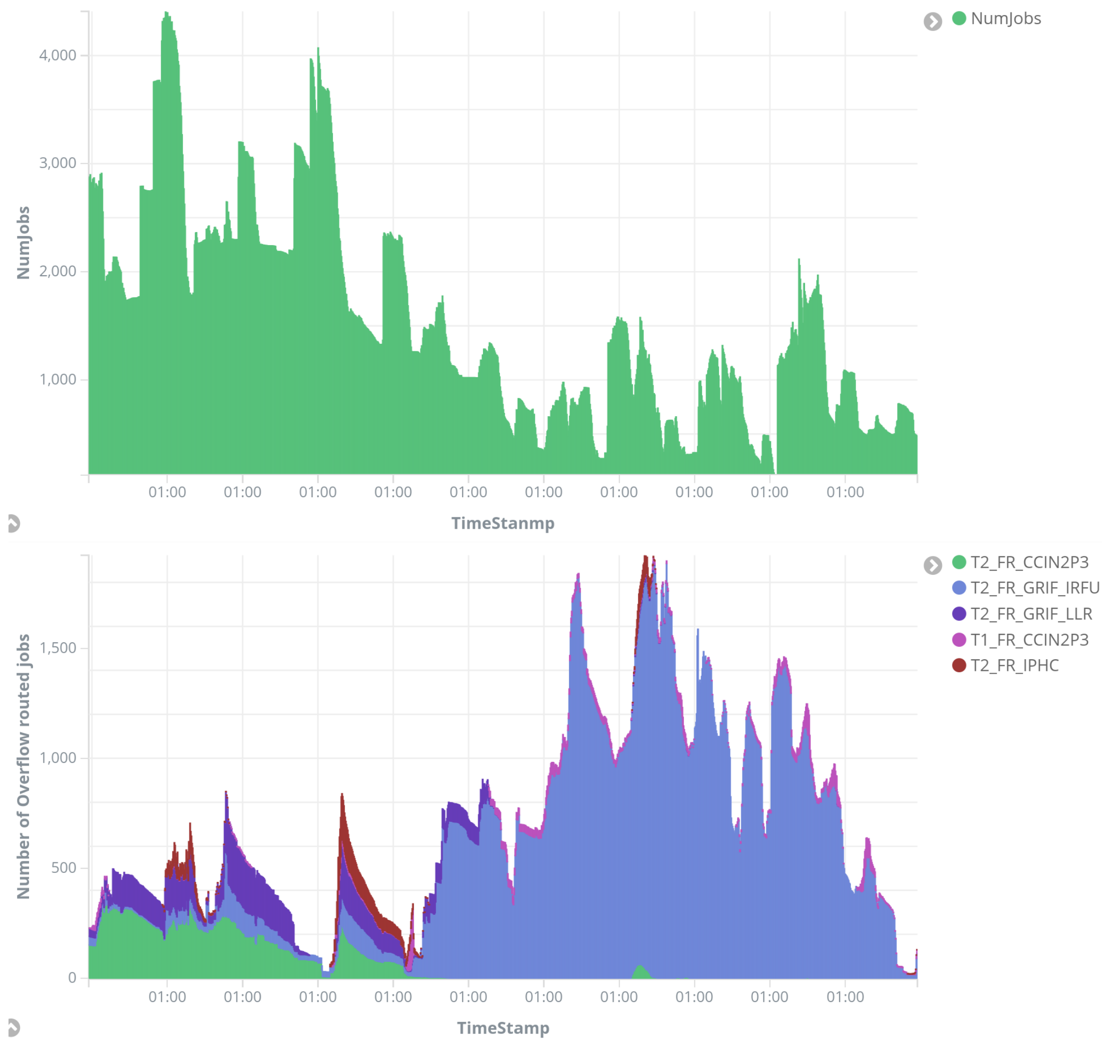


Figure 6: Top: number of idle jobs at the French tier-1 site. Bottom: number of jobs overflown from Tier-1 to Tier-2 sites in France.

the number of running jobs at French Tier-2 sites which have been overflown. A clear correlation between the trends in the two plots is evident, the second being shifted in time with respect to the first because of infrastructure delay.

4. Conclusions and directions for further work

The goal of CRAB is to provide CMS researchers with the ability to exploit the distributed worldwide resources of the worldwide LHC computing grid to execute analysis tasks ranging from simple to massive in scope. Complex tasks need to be split into individual jobs that can be run in the few hundred thousand CPU cores available worldwide. 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

PoS(ISGC2019)015

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 analytics.

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] T. Ivannov et al., "Improving efficiency of analysis jobs in CMS", in press.

[3] I. Sfiligoi et al., "The Pilot way to grid resources using GlideinWMS", 2009 WRI World Congress on Computer Science and Information Engineering, doi 10.1109/CSIE.2009.950.

[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. https://research.cs.wisc.edu/htcondor/index.html

[5] J. Letts et.al., "Improving the scheduling efficiency of a global multi-core HTCondor pool in CMS", in press.

[6] A. Perez-Calero Yzquierdo et al. "Exploring GlideinWMS and HTCondor scalability frontiers for an expanding CMS global pool", in press.

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

[8] J. Balcas et al., "HTTP as a data access protocol: trials with XrootD in CMS?s AAA project", 2017 J. Phys.: Conf. Ser. **898** 062042.

PoS(ISGC2019)015