EPJ Web of Conferences **214**, 03053 (2019) https://doi.org/10.1051/epjconf/201921403053
CHEP 2018

Adoption of ARC-CE and HTCondor at GridKa Tier 1

Max Fischer[1,*], *Eileen* Kuehn[1,**], *Matthias* Schnepf[1,***], *Andreas* Petzold[1,****], and *Andreas* Heiss[1,†]

[1]Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Abstract. The GridKa Tier 1 data and computing center hosts a significant share of WLCG processing resources. Providing these resources to all major LHC and other VOs requires efficient, scalable and reliable cluster management. To satisfy this, GridKa has recently migrated its batch resources from CREAM-CE and PBS to ARC-CE and HTCondor. This contribution discusses the key highlights of the adoption of this middleware at the scale of a European Tier 1 center: As the largest WLCG Tier 1 using the ARC-CE plus HTCondor stack, GridKa is exemplary for migrating more than 20 000 cores over the time span of only a few weeks. Supporting multiple VOs, we have extensively studied the constraints and possibilities of scheduling jobs of vastly different requirements. We present a robust and maintainable optimization of resource utilization which still respects constraints desired by VOs. Furthermore, we explore the dynamic extension of our batch system, integrating cloud resources with a lightweight configuration mechanism.

1 Introduction

The German GridKa Computing Centre [1] is amongst the largest Tier 1 centers in the Worldwide LHC Computing Grid (WLCG) [2]. At the start of 2018, the entire GridKa processing resources were migrated from CREAM-CE and PBS to ARC [3] Compute Element and HTCondor [4] Batch System. This involved the migration of more than 20,000 CPU cores, as well as new policies for both the Compute Element and Batch System. While the migration itself is of course technically similar to previous work [5], we have used the opportunity to investigate the challenges of WLCG sites in a controlled environment and at a larger scope.

Supporting all major collaborations active in the WLCG, GridKa is exemplary for handling a wide range of workflows. This includes both a variance in resource requests, as well as peculiarities of the various frameworks in use. Near-complete utilization of the currently almost 30 000 CPU cores requires both scalable and manageable scheduling policies. To this end, we have investigated key features of WLCG workflows for scheduling, reviewed the implications and countermeasures for resource fragmentation, and devised a novel yet simple approach to precisely control fragmentation.

*e-mail: max.fischer@kit.edu
**e-mail: eileen.kuehn@kit.edu
***e-mail: matthias.schnepf@kit.edu
****e-mail: andreas.petzold@kit.edu
†e-mail: andreas.heiss@kit.edu

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (http://creativecommons.org/licenses/by/4.0/).

2 Scheduling Challenges for WLCG Sites

Sites of the WLCG are faced with distinct challenges to schedule the workflows of LHC collaborations. Individual resource requests are significantly smaller than the capacity of worker nodes, preventing whole-node scheduling as used by HPC sites. Yet the total processing resources are not significantly larger than the total resource requests, requiring full utilization of all available resources. As a result, scheduling must be optimized for both small-scale and large-scale features of requests.

We have analyzed job features based on a sample of jobs from all major LHC collaborations, namely ALICE [6], ATLAS [7], CMS [8] and LHCb [9], submitted during June 2018. Submission frequency, request size and observed runtime are shown in Figures 1, 2 and 3, respectively. Note that this data serves to demonstrate fundamental features for scheduling - it is not meant to represent or quantify features of WLCG jobs in general.

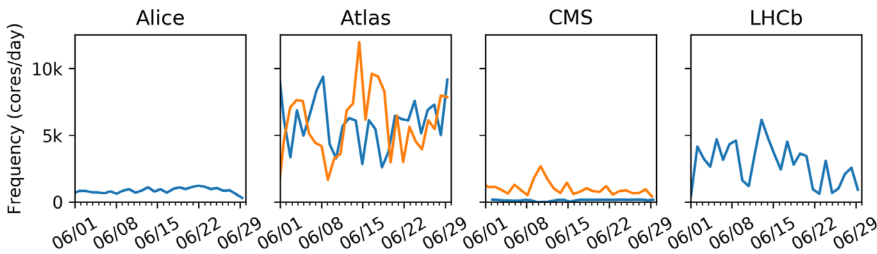

Figure 1. Frequency of jobs: Volume of job requests per day to the GridKa batch system. Each job is counted for the day it entered the batch system. The total number of cores requested is shown, in order to reflect the total resources in each category. Blue lines are 1-core requests, orange lines are 8-core requests. Jobs submitted with an explicit service or test identity are excluded; however, service and test jobs using a regular identity cannot be excluded.

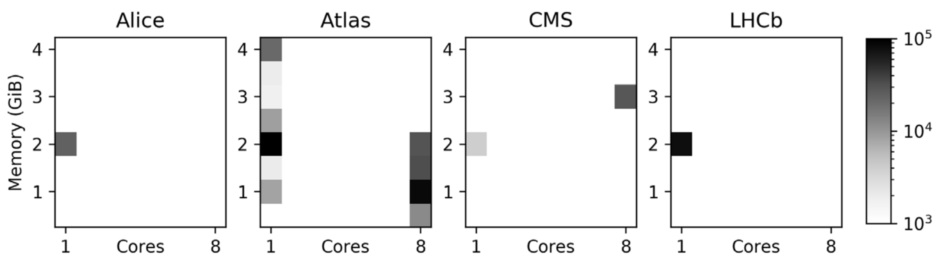

Figure 2. Size of jobs: Resources requested by jobs submitted to GridKa. Memory is the amount of RAM requested per core. Saturation represents the total number of cores requested, in order to reflect the total resources in each category. Data covers the same jobs as Figure 1.

2.1 Multi-VO fairshare

Each collaboration served by the WLCG is represented by a so-called virtual organization (VO). In turn, each VO is entitled to use a share of resources per WLCG site, reflecting the resources pledged by the size to support the collaboration. Consequently, every job on a

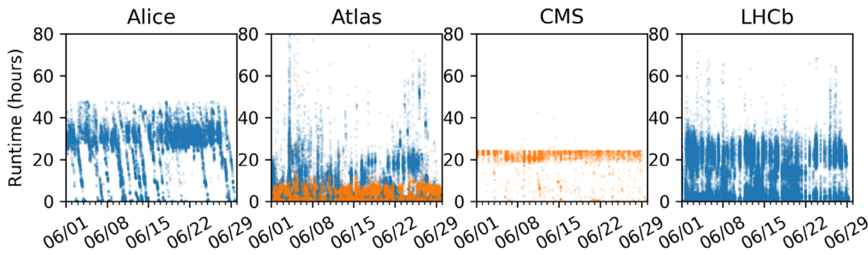

Figure 3. Runtime of jobs: Walltime of jobs run by the GridKa batch system. Each job is counted for the day it entered the batch system. Saturation represents the count of jobs per area. Blue dots are 1-core requests, orange dots are 8-core requests. Data covers the same jobs as Figure 1, with the exception of jobs shorter than 120 seconds.

WLCG site is attributed to a VO; the total resources used by the jobs of a VO should match its share.

As shown in Figure 1, the resources requested by each VO are not constant over time. Variations between minimum and maximum requests are on the order of factor 2 to 5. To avoid idle resources, shares must not be statically allocated to VOs: Unused resources must be allocated to other VOs, and short-term share must compensate long-term under- or over-utilization from this.

2.2 Partitioning for multi-size job demand

Jobs at GridKa may use either 1 or 8 cores, yet may freely request memory up to 4 GiB per core. Resource requests cover various sizes in terms of memory and cores, as shown in Figure 2. Notably, sizes vary both across and inside VOs. Every VO uses requests for 1-core, while only some use requests for 8-cores. Custom memory sizes are requested by only one VO.

Since worker nodes can satisfy multiple requests, resources must be partitioned to match request sizes. As request sizes as well as total requests per VO may vary, this requires a dynamic partitioning depending on demand. However, resource requests require contiguous resources and thus requests of different size cannot be treated equally: a small request can always take the place of a larger request, but not vice versa.

2.3 Late-Bound Pilot jobs

Jobs at GridKa are almost entirely pilots [10] - placeholders that pull in actual jobs once executing on a worker node. This late-binding precludes knowledge about pilot lifetime, as pilots may terminate early if no jobs are available [11]. Since the late-binding strategies vary between VOs, the lifetimes of jobs also vary by VO, as shown in Figure 3.

Different lifetimes of jobs imply different weight to satisfy the resource share over time. For example, given jobs of full and half lifetime, the latter must be started twice as often. As a result, every scheduling decision must reflect the long-term effects over time.

3 Dynamic Partitioning and Emergent Fragmentation

The challenges outlined in Section 2 necessitate dynamic allocation and partitioning of resources. Changes in frequency, size and weight of resource requests make it necessary to

constantly adjust the size of individual resource allocations. To this end, HTCondor provides the PartitionableSlot [12] mechanism: a large chunk of resources may be partitioned into smaller chunks, each matching an individual job. While this mechanism allows to easily match current demand, matching long-term demand requires non-trivial policies.

Each partitionable slot represents a pool of contiguous resources, such as an entire worker node. A job matched to such a slot temporarily removes its requested resources from the pool, returning them after completion. For example, this allows a 32 core slot to serve either 32 1-core jobs, 4 8-core jobs, or 2 8-core and 16 1-core jobs without reconfiguration. However, partitionable slots under high utilization are subject to *Fragmentation*, which prevents larger jobs from starting.

3.1 Partitionable Slot Fragmentation

Even though a partitionable slot may represent a sizeable pool of resources, at any moment it can only provide the resources not used by running jobs. Under high utilization, this means resources for new jobs are only available from jobs that recently finished and thus released their resources. As a result, only jobs can start that request equal or less resources than recently released. Fragmentation describes the state where certain classes of jobs cannot start, because there are not enough resources released at once.

This effect implicitly discriminates jobs requesting many resources at once: the more resources a job requests, the more jobs can compete for the required resources. As a result, there is an ordering of competing jobs depending on resource requests, see Figure 4. This ordering causes fragmentation to emerge naturally: large partitions gradually degrade to smaller partitions.

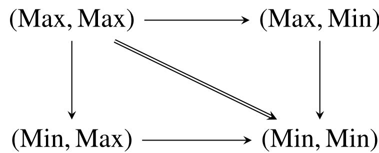

Figure 4. Competition of jobs for resources of partitionable slots, exemplary for two resource types such as (CPU, RAM). Maximum requests compete against all others, while minimum requests compete only against themselves. Imbalanced requests only compete against a subset of requests.

We have observed the discriminating effects of this implicit ordering in various aspects of scheduling. Policies that are stable under ideal conditions may disproportionately favor small jobs overall. Since it is natural for resources to fragment, but not to defragment, there is a bias that must be counterbalanced.

3.2 Small Job Handicap

Degradation can be directly countered by actively discriminating small jobs in scheduling. We have tried two approaches, also employed at other WLCG sites [13]:

1. Separating large and small jobs into different scheduling sub-groups. For example, a VO Bob may be represented by the groups Bob.SC and Bob.MC for 1-core and 8-core requests. This allows scheduling large jobs first[1], before small jobs may degrade free resources.

[1] Using the GROUP_SORT_EXPR setting of HTCondor.

2. Applying a grace period on large resource chunks before they are available for smaller requests. For example, a worker node with more than 8 cores may reject 1-core jobs for some time. This reserves chunks of resources for large jobs, running small jobs only as a fallback.

While both of these methods successfully counteract the degradation of resources, we have found them not robust in our conditions. Approach 1) prevents small jobs from running even when there are only few large jobs. The *static* priority proved inappropriate to reflect the *variable* demand for small and large jobs per VO share (see Figure 1). In contrast, approach 2) is vulnerable against a high turnaround of jobs. Jobs with short duration excessively trigger grace periods, leading to unavailable and thus underutilized resources. An optimal grace period could not be defined, given the variance of job runtimes (see Figure 3). Both approach 1) and 2) failed to preserve the fair-share between VOs due to their inherent bias.

3.3 Remainder Scheduling

The default HTCondor policy fills individual partitionable slots completely by preferring slots with the least remaining cores (Slot.CPUS) and memory (Slot.RAM) (see Equation 1). This ensures that the free resources are concentrated onto few nodes, providing large contiguous chunks. While a good fit for the general case, it is not optimal if jobs are restricted to discrete resource volumes.

In our case jobs are restricted to request 1- or 8-cores, and between 2GB to 4GB of RAM per core (see Figure 2). This means there is no advantage creating contiguous chunks which are not multiples of 8-cores, or which have less than 2GB or more than 4GB of RAM per core. For example, matching a 1-core job against either an 8-core slot and a 9-core slot, we may choose the 9-core slot without destroying a chunk. However, the default policy consistently chooses the 8-core slot, destroying the chunk.

In order to preserve suitably sized chunks only, we have chosen a policy based on resources remaining *after* a potential match. Given that the desired largest chunk size is known, we can calculate the chunks preserved by a match based on requested cores (Job.CPUS) and memory (Job.RAM) (see Equation 2). Instead of forming arbitrarily large chunks as a side-effect, our approach forms suitably large chunks on purpose. The edge case of perfect fit without remainder is special cased (see Equation 3) which also simplifies remainder calculation itself.

$$\mathrm{RANK}_{\mathrm{def}} \propto -\mathrm{Slot.CPUS} - \mathrm{Slot.RAM} \tag{1}$$

$$\mathrm{RANK}_{\mathrm{rem}} \propto \mathrm{floor}\left(\frac{\mathrm{Slot.CPUS} - \mathrm{Job.CPUS}}{8}\right) + \left(2000 < \frac{\mathrm{Slot.RAM} - \mathrm{Job.RAM}}{\mathrm{Slot.CPUS} - \mathrm{Job.CPUS}}\right) \tag{2}$$

$$\mathrm{RANK}_{\mathrm{fit}} \propto (\mathrm{Slot.CPUS} == \mathrm{Job.CPUS}) + (\mathrm{Slot.RAM} == \mathrm{Job.RAM}) \tag{3}$$

3.4 Active Defragmentation

The optional HTCondor DEFRAG daemon [12] has the sole purpose of actively removing fragmentation. This is achieved by selecting fragmented worker nodes, and forbidding new jobs until sufficient resources are free. What constitutes a fragmented worker node and sufficient resources is freely adjustable. The approaches we have tried fall into two categories:

1. Absolute defragmentation that aims at even global fragmentation across all worker nodes. This relies on the manual derivation of optimal *limits for fragmentation*. Practically, each worker node must have several large chunks, regardless whether they are currently filled or not.

2. Relative defragmentation that aims at gradually removing local fragmentation. This relies on the manual derivation of optimal *speed of fragmentation*. Practically, a few worker nodes must have one large chunk that is currently not filled.

While approach 1) works well with small worker nodes, we have found it unsuitable when more than two chunks are required per node. Since Defrag can only disable starting of all jobs, we have repeatedly observed idle large slots. This occurs when a chunk is missing, but small jobs are taking long to release their resources. Until the small jobs are finished, large jobs are not replaced either.

In contrast, approach 2) works well for large worker nodes that may need to provide several large chunks. Since Defrag then only aims at one free large chunk a large job finishing immediately satisfies this, aborting defragmentation and allowing replacement. This implicitly makes the defragmentation speed dependent on the fragmentation of the cluster: as the number of large jobs increases the chance of aborting defragmentation due to large jobs finishing increases as well. We have observed a practical limit for relative defragmentation, when there is always a large job that finishes before small jobs free a new chunk.

4 Using Concurrency Limits to manage Fragmentation

The viable countermeasures against fragmentation fall into two categories: node-scale measures that rely on hard limits, and cluster-scale measures that rely on soft priorities. This makes it difficult to reliably manage fragmentation when demand changes significantly in the whole cluster (see Section 2). As a result, we desire a hard limit at the scale of our entire cluster.

We have found that our mechanism used to limit jobs per VO can also be used to limit fragmentation. This is related to managing fragmentation via scheduling priorities, but imposes absolute limits. However, this mechanism only supplies the means to statically limit fragmentation; therefore, we have also developed an approach to dynamically adjust this mechanism.

4.1 Concurrency Tagging and Limits

The HTCondor mechanism ConcurrencyLimits allows to set cluster-wide limits on arbitrary, countable resources. This requires a job to declare how many resources of a given type it blocks. If a concurrency limit is defined for this resource type, the HTCondor Negotiator ensures that the sum of blocked resources does not exceed the limit. This is similar to how resources of a worker node are allocated, but applies to the whole cluster. Effectively, this means there is a global limit to how many jobs using a specific resource may run at once.

We have modified our Compute Elements to automatically tag jobs as using pseudo-resources that represent job sizes. For example, 8 core jobs are tagged to use the resources `slots.mc:1,cores.mc:8`, i.e. one large slot and 8 cores of a large slot. We currently categorize all congested resources into small and large, i.e. large/small core chunk and large/small memory chunk. Individual categories are orthogonal, i.e. a job may use a large core chunk with a small memory chunk.

This tagging scheme allows setting a global, hard limit on fragmentation. For example, setting `slots.sc_LIMIT = 14000` prevents more than 14 000 small jobs from running – thereby saving the remaining resources for larger jobs. Notably, this approach does not ensure that free resources are adequately compacted; a compacting scheduling priority, as outlined in Section 3.3, should be employed as well.

4.2 Adjusting Limits as Opportunistic Resources

Using ConcurrencyLimits to manage fragmentation effectively partitions a cluster. This is effective for guarding against unbounded fragmentation, but not suitable for precisely matching the needs of jobs. Since limits are static, this partitioning must be adjusted from the outside to match demand. This challenge is related to the ongoing work on opportunistic resources: the amount of required resources must be deduced based on job needs.

We have implemented a prototype to manage ConcurrencyLimits with COBalD [14], our framework to manage opportunistic resources based on observed demand. This mechanism reserves a fraction of the cluster for *large* jobs by managing a limit on *small* jobs. In case reserved resources are not utilized, e.g. due to a lack of demand, the limit is relaxed to ensure high utilization. This allows us to manage large jobs by controlling the absence of small jobs, as shown in Figure 5.

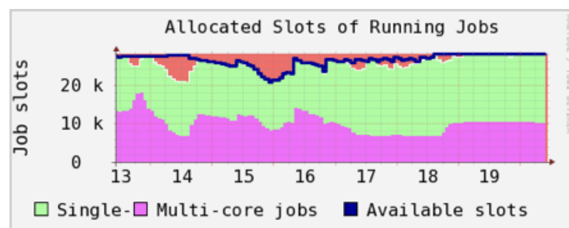

Figure 5. Allocated slots for 1-core (single) and 8-core (multi) jobs at GridKa while using ConcurrencyLimits to manage fragmentation. Until day 16, a static limit for 1-core jobs is used; the underutilization at day 14 is due to a lack of 8-core jobs and the limit preventing 1-core jobs as a fallback. Following day 16, the limit is managed dynamically as an opportunistic resource. Underutilization only occurs when 1-core slots are compacted to free larger chunks.

5 Conclusion

Modern Middleware makes it technically feasible to migrate entire clusters of thousands of cores from one technology to another. Managing resources at the scale of WLCG Sites is an increasing challenge due to constantly rising volumes of required resources. While the technical challenges have become manageable, the scale of operations requires optimal policies to effectively utilize available resources.

An increasing problem is the fragmentation of resources, a natural result of the dynamic resource demands in the WLCG. As the size of individual worker nodes and entire clusters grows, allocation becomes increasingly biased towards smaller jobs. This requires active countermeasures, which are themselves subject to effects from scaling up available resources. We have surveyed and tested several methods commonly employed to manage fragmentation in HTCondor clusters. While several of these have proven effective at slowing down fragmentation, none offers precise control at the scope of an entire cluster.

As a response, we have constructed a new approach for fragmentation control: by tagging jobs on submission, we can apply cluster-wide, hard limits for jobs that cause fragmentation. Combining this approach with previous work on managing opportunistic resources, we have successfully achieved the means to control fragmentation with regard to resource demands.

References

[1] KIT Site Report 2017 **URL** https://indico.cern.ch/event/637013/contributions/2739322/ [accessed 2010-02-04]

[2] J. Shiers "The Worldwide LHC Computing Grid (worldwide LCG)" *Computer Physics Communications* **177** 219–223 (2007)

[3] The NorduGrid project homepage **URL** http://www.nordugrid.org [accessed 2018-12-01]

[4] Center for High Throughput Computing. (2018, March 13). HTCondor 8.7.7 (Version 8.7.7). Zenodo. http://doi.org/10.5281/zenodo.1206284

[5] S. U. Ahn, A. Jaikar, B. Kong, I. Yeo, S. Bae, J. Kim "Experience on HTCondor batch system for HEP and other research fields at KISTI-GSDC" *J. Phys.: Conf. Ser.* **898** 082013 (2017)

[6] The ALICE Collaboration "The ALICE heavy-ion experiment at the CERN LHC" *Nuclear Physics A* **566** 311-319 (1994)

[7] The ATLAS Collaboration "The ATLAS Experiment at the CERN Large Hadron Collider" *J. Instrum.* **3** S08003 (2008)

[8] The CMS Collaboration "The CMS experiment at the CERN LHC" *J. Instrum.* **3** S08004 (2008)

[9] The LHCb Collaboration "The LHCb Detector at the LHC" *J. Instrum.* **3** S08005 (2008)

[10] M. Turilli, M. Santcroos, S. Jha "A Comprehensive Perspective on Pilot-Job Systems" *ACM Comput. Surv.* **51** 43 (2018) https://doi.org/10.1145/3177851

[11] P. A. Love, M. Alef, S. Dal Pra, A. Di Girolamo, A. Forti, J. Templon, E. Vamvakopoulos, the ATLAS Collaboration "Analysis of empty ATLAS pilot jobs"" *J. Phys.: Conf. Ser.* **898** 092005 (2017)

[12] The HTCondor Manual, Section 3.7, Policy Configuration for Execute Hosts and for Submit Hosts **URL** http://research.cs.wisc.edu/htcondor/manual/v8.6/3_7Policy_Configuration.html [accessed 2019-02-04]

[13] The GridPP wiki, Example Build of an ARC/Condor Cluster **URL** https://www.gridpp.ac.uk/wiki/Example_Build_of_an_ARC/Condor_Cluster [accessed 2018-12-01]

[14] M. Fischer, E. Kuehn, M. Giffels. (2018, December 3). MaineKuehn/cobald: Working version for HNSC and ConcurrencyLimits (Version v0.9.1). Zenodo. http://doi.org/10.5281/zenodo.1887873